Procedural Relection in Programming Languages

3a Preliminaries

1 Abstract†

We show how a computational system can be constructed to "reason," effectively and consequentially, about its own inferential processes. The analysis proceeds in two parts. First, we consider the general question of computational semantics, rejecting traditional approaches, and arguing that the *declarative* and *procedural* aspects of computational symbols (what they stand for, and what behaviour they engender) should be analysed *independently*, in order that they may be coherently related. Second, we investigate *self-referential* behaviour in computational processes, and show how to embed an effective procedural model of a computational calculus within that calculus (a model not unlike a meta-circular interpreter, but connected to the fundamental operations of the machine in such a way as to provide, at any point in a computation, fully articulated descriptions of the state of that computation, for inspection and possible modification). In terms of the theories that result from these investigations, we present a general architecture for **procedurally reflective** processes, able to shift smoothly between dealing with a given subject domain, and dealing with their own reasoning processes over that domain.

An instance of the general solution is worked out in the context of an applicative language. Specifically, we present three successive dialects of Lisp: **1Lisp**,‡ a distillation of current practice, for comparison purposes; **2Lisp**, a dialect constructed in terms of our rationalised semantics, in which the

†The section numbers used here ('1' for the Abstract; '2', Extended Abstract; '3', Preface; and '4', Prologue) were introduced for this version.

‡As indicated in the Cover, in this version I have removed the hyphens from the labels '1-Lisp,' '2-Lisp,' and '3-Lisp' used in the dissertation, thus using '1Lisp,' '2Lisp,' and '3Lisp,' respectively.

concept of evaluation is rejected in favour of independent notions of *simplification* and *reference*, and in which the respective categories of notation, structure, semantics, and behaviour are strictly aligned; and **3Lisp**, an extension of 2Lisp endowed with reflective powers.

2 Extended Abstract

We show how a computational system can be constructed to "reason" effectively and consequentially about its own inference processes. Our approach is to analyse *self-referential* behaviour in computational systems, and to propose a theory of *procedural reflection* that enables any programming language to be extended in such a way as to support programs able to access and manipulate structural descriptions of their own operations and structures. In particular, one must encode an explicit theory of such a system within the structures of the system, and then connect that theory to the fundamental operations of the system in such a way as to support three primitive behaviours. First, at any point in the course of a computation, fully articulated descriptions of the state of the reasoning process must be available for inspection and modification. Second, it must be possible at any point to resume an arbitrary computation in accord with such (possibly modified) theory-relative descriptions. Third, procedures that reason with descriptions of the processor state must themselves be subject to description and review, to arbitrary depth. Such *reflective* abilities allow a process to shift smoothly between dealing with a given subject domain, and dealing with its own reasoning processes over that domain.

Crucial in the development of this theory is a comparison of the respective semantics of programming languages (such as Lisp and Algol) and declarative languages (such as logic and the λ-calculus); we argue that unifying these traditionally separate disciplines clarifies both, and suggests a simple

and natural approach to the question of procedural reflection. More specifically, the semantical analysis of computational systems should comprise independent formulations of **declarative import** (what symbols stand for) and **procedural consequence** (what effects and results are engendered by processing them), although the two semantical treatments may, because of side-effect interactions, have to be formulated in conjunction. When this approach is applied to a functional language it is shown that the traditional notion of *evaluation* is confusing and confused, and must be rejected in favour of independent notions of *reference* and *simplification*. In addition, we defend a standard of **category alignment**: there should be a systematic correspondence between the respective categories of notation, abstract structure, declarative semantics, and procedural consequence (a mandate satisfied by no extant procedural formalism). It is shown how a clarification of these prior semantical and aesthetic issues enables a *procedurally reflective* dialect to be clearly defined and readily constructed.

An instance of the general solution is worked out in the context of an applicative language, where the question reduces to one of defining an interpreted calculus able to inspect and affect its own interpretation. In particular, we consider three successive dialects of Lisp: 1Lisp, a distillation of current practice for comparison purposes; 2Lisp, a dialect *categorically* and *semantically rationalised* with respect to an explicit theory of declarative semantics for s-expressions; and 3Lisp, a derivative of 2Lisp endowed with full reflective powers. 1Lisp, like all Lisp dialects in current use, is at heart a *first-order* language, employing meta-syntactic facilities and dynamic variable scoping protocols to partially mimic higher-order functionality. 2Lisp like Scheme and the λ-calculus, is higher-order: it supports arbitrary function designators in argument position, is lexically scoped, and treats the function position of an application in a standard extensional manner. Unlike Scheme, however, the

2Lisp processor is based on a regimen of *normalisation*, taking each expression into a normal-form co-designator of its referent, where the notion of *normal-form* is in part defined with respect to that referent's semantic type, not (as in the case of the λ-calculus) solely in terms of the further non-applicability of a set of syntactic reduction rules. 2Lisp normal-form designators are environment-independent and side-effect free; thus the concept of a *closure* can be reconstructed as a *normal-form function designator*. In addition, since normalisation is a form of simplification, and is therefore *designation-preserving*, meta-structural expressions are not de-referenced upon normalisation, as they are when evaluated. Thus we say that the 2Lisp processor is **semantically flat**, since it stays at a semantically fixed level (although explicit referencing and de-referencing primitives are also provided, to facilitate explicit level shifts). Finally, because of its category alignment, *argument objectification* (the ability to apply functions to a sequence of arguments designated collectively by a single term) can be treated in the 2Lisp base-level language, without requiring resort to meta-structural machinery.

3Lisp is straightforwardly defined as an extension of 2Lisp, with respect to an explicitly articulated procedural theory of 3Lisp embedded in 3Lisp structures. This embedded theory, called the **reflective model**, though superficially resembling a meta-circular interpreter, is causally connected to the workings of the underlying calculus in crucial and primitive ways. Specifically, *reflective procedures* are supported that bind as arguments (designators of) the continuation and environment structure of the processor that would have been in effect at the moment the reflective procedure was called, had the machine been running all along in virtue of the explicit processing of that reflective model. Because reflection may recurse arbitrarily, 3Lisp is most simply defined as an infinite tower of 3Lisp processes, each engendering the process immediately below

it. Under such an account, the use of reflective procedures amounts to running programs at arbitrary levels in this reflective hierarchy. Both a straightforward implementation and a conceptual analysis are provided to demonstrate that such a machine is nevertheless finite.

The 3Lisp reflective model unifies three programming language concepts that have formerly been viewed as independent: meta-circular interpreters, explicit names for the primitive interpretive procedures (EVAL and APPLY in standard Lisp dialects), and procedures that access the state of the implementation (typically provided, as part of a programming environment, for debugging purposes). We show how all such behaviours can be defined within a pure version of 3Lisp (i.e., independent of implementation), since all aspects of the state of any 3Lisp process are available, with sufficient reflection, as objectified entities within the 3Lisp **structural field**.

3 Preface

The possibility of constructing a reflective calculus first struck me in June 1976, at the Xerox Palo Alto Research Center (PARC), where I was spending a summer working with the KRL representation language of Bobrow and Winograd.[1] As an exercise to learn the new language, I had embarked on the project of representing KRL in KRL; it seemed to me that this "double-barreled" approach, in which I would have both to *use* and to *mention* the language, would be a particularly efficient way to unravel its intricacies. Though that exercise was ultimately abandoned, I stayed with it long enough to become intrigued by the thought that one might build a system that was self-descriptive in an important way (certainly in a way in which my KRL project was *not*). More specifically, I could dimly envisage a computational system in which what happened took effect in virtue of declarative descriptions of what was to

[1] 'KRL' for 'Knowledge Representation Language; see Bobrow and Winograd (1977) and Bobrow et al. (1977).

happen, and in which the internal structural conditions were represented in declarative descriptions of those internal structural conditions. In such a system a program could with equal ease access all the basic operations and structures either directly or in terms of completely (and automatically) articulated descriptions of them. The idea seemed to me rather simple (as it still does); furthermore, for a variety of reasons I thought that such a reflective calculus could *itself* be rather simple—in some important ways simpler than a non-reflective formalism (this too I still believe). *Designing* such a formalism, however, no longer seems as straightforward as I thought at the time; this dissertation should be viewed as the first report emerging from the research project that ensued.

Most of the five years since 1976 have been devoted to initial versions of my specification of such a language, called **Mantiq**, based on these original hunches. As mentioned in the A1
first paragraph of [dissertation] chapter 1,† there are various A2
non-trivial goals that must be met by the designer of any such formalism, including at least a tentative solution to the knowledge representation problem. Furthermore, in the course of its development, Mantiq has come to rest on some additional hypotheses above and beyond those mentioned above (including, for example, a sense that it will be possible within a computational setting to construct a formalism in which syntactic identity and intensional identity can be identified, given some appropriate, but *independently* specified, theory of intensionality). Probably the major portion of my attention to date has A3
focused on these intensional aspects of the Mantiq architecture.

It was clear from the outset that no dialect of Lisp (or of any other purely procedural calculus) could serve as a full reflective formalism; purely declarative languages like logic or the λ-calculus were dismissed for similar reasons. In Febru- A4
ary of 1981, however, I decided that it would be worth focus-

†Included here as ch. 3b, p.+.

ing on Lisp, by way of an example, in order to work out the details of a specific subset of the issues with which Mantiq would have to contend. In particular, I recognised that many of the questions of reflection could be profitably studied in a (limited) procedural dialect, in ways that would ultimately illuminate the larger programme. Furthermore, to the extent that Lisp could serve as a theoretical vehicle, it seemed a good project; it would be much easier to develop, and even more so to communicate, solutions in a formalism at least partially A5 understood.

The time from the original decision to look at procedural reflection (and its concomitant emphasis on semantics—I realised from investigations of Mantiq that semantics would come to the fore in all aspects of the overall enterprise), to a working implementation of 3Lisp, was only a few weeks. Articulating why 3Lisp was the way it was, however—i.e., formulating in plain English the concepts and categories on which the design was founded—required quite intensive work for the remainder of the year. A first draft of the dissertation was completed at the end of December 1981; the implementation remained essentially unchanged during the course of this writing (the only substantive alteration was the idea of treating recursion in terms of explicit Y-operators). Thus—and I suspect there is nothing unusual in this experience—formulating an idea required approximately ten times more work than embodying it in a machine; perhaps more surprisingly, all of that effort in formulation occurred *after* the implementation was complete. We sometimes hear that writing computer programs is intellectually hygienic because it requires that we make our ideas completely explicit. I have come to disagree rather fundamentally with this view. Certainly writing a pro- A6 gram does not force one to one make one's ideas *articulate*, although it is a useful first step. More seriously, however, it is often the case that the organising principles and fundamental

insights contributing to the coherence of a program are not explicitly encoded within the structures comprising that program. The theory of declarative semantics embodied in 3Lisp, for example, was initially tacit—a fact perhaps to be expected, since only procedural consequence is explicitly encoded in an implementation. Curiously, this is one of the reasons that building a fully reflective formalism (as opposed to the limited procedurally reflective languages considered here) is difficult: in order to build a general reflective calculus, one must embed within it a fully articulated theory of one's understanding of it. This will take some time.

4 Prologue

It is a striking fact about human cognition that we can think not only about the world around us, but also about our ideas, our actions, our feelings, our past experience. This ability to **reflect** lies behind much of the subtlety and flexibility with which we deal with the world; it is an essential part of mastering new skills, of reacting to unexpected circumstances, of short-range and long-range planning, of recovering from mistakes, of extrapolating from past experience, and so on and so forth. Reflective thinking characterises mundane practical matters and delicate theoretical distinctions. We have all paused to review past circumstances, such as conversations with guests or strangers, to consider the appropriateness of our behaviour. We can remember times when we stopped and consciously decided to consider a set of options, say when confronted with a fire or other emergency. We understand when someone tells us to believe everything a friend tells us, unless we know otherwise. In the course of philosophical discussion we can agree to distinguish views we believe to be true from those we have no reason to believe are false. In all these cases the subject matter of our contemplation at the moment of reflection includes our remembered experience, our private thoughts, and our reasoning patterns.

The power and universality of reflective thinking has caught the attention of the cognitive science community—indeed, once alerted to this aspect of human behaviour, theorists find evidence of it almost everywhere. Though no one can yet say just what it comes to, crucial ingredients would seem to be the ability to recall memories of a world experienced in the past and of one's own participation in that world, the ability to think about a phenomenal world, hypothetical or actual, that is not currently being experienced (an ability presumably mediated by our knowledge and belief), and a certain kind of true self-reference: the ability to consider both one's actions and the workings of one's own mind. This last aspect—the self-referential aspect of reflective thought—has sparked particular interest for cognitive theorists, both in psychology (under the label *meta-cognition)* and in artificial intelligence (in the design of computational systems possessing inchoate reflective powers, particularly as evidenced in a collection of ideas loosely allied in their use of the term "meta": meta-level rules, meta-descriptions, and so forth).

In artificial intelligence, the focus on computational forms of self-referential reflective reasoning has become particularly central. Although the task of endowing computational systems with subtlety and flexibility has proved difficult, we have had some success in developing systems with a moderate grasp of certain domains: electronics, bacteremia, simple mechanical systems, etc. One of the most recalcitrant problems, however, has been that of developing flexibility and modularity (in some cases even simple effectiveness) in the reasoning processes that use this world knowledge. Though it has been possible to construct programs that perform a specific kind of reasoning task (say, checking a circuit or parsing a subset of natural language syntax), there has been less success in simulating "common sense," or in developing programs able to figure out what to do, and how to do it, in either general or novel situations. If the

course of reasoning—if the problem solving strategies and the hypothesis formation behaviour—could *itself* be treated as a valid subject domain in its own right, then (at least so the idea goes) it might be possible to construct systems that manifested the same modularity about their own thought processes that they manifest about their primary subject domains. A simple example might be an electronics "expert" able to choose an appropriate method of tackling a particular circuit, depending on a variety of questions about the relationship between its own capacities and the problem at hand: whether the task was primarily one of design or analysis or repair, what strategies and skills it knew it had in such areas, how confident it was in the relevance of specific approaches based on, say, the complexity of the circuit, or on how similar it looked compared with circuits it already knew. Expert human problem-solvers clearly demonstrate such reflective abilities, and it appears more and more certain that powerful computational problem solvers will have to possess them as well.

No one would expect potent skills to arise automatically in a reflective system; the mere *ability* to reason about the reasoning process will not magically yield systems able to reflect in powerful and flexible ways. On the other hand, the demonstration of such an ability is clearly a pre-requisite to its effective utilisation. Furthermore, many reasons are advanced in support of reflection, as well as the primary one (the hope of building a system able to decide how to structure the pattern of its own reasoning). It has been argued, for example, that it would be easier to construct powerful systems in the first place (it would seem you could almost *tell them* how to think), to interact with them when they fail, to trust them if they could report on how they arrive at their decisions, to give them "advice" about how to improve or discriminate, as well as to provide them with their own strategies for reacting to their history and experience.

There is even, as part of the general excitement, a tentative suggestion on how such a self-referential reflective process might be constructed. This suggestion—nowhere argued but clearly in evidence in several recent proposals—is a particular instance of a general hypothesis, adopted by most A.I. researchers, that we will call the *Knowledge Representation Hypothesis*. It is widely held in computational circles that any process capable of reasoning intelligently about the world must consist in part of a field of structures, of a roughly linguistic sort, which in some fashion *represent* whatever knowledge and beliefs the process may be said to possess. For example, according to this view, since I know that the sun sets each evening, my "mind" must contain (among other things) a language-like or symbolic structure that represents this fact, inscribed in some kind of internal code. There are various assumptions that go along with this view: there is for one thing presumed to be an internal process that "runs over" or "computes with" these representational structures, in such a way that the intelligent behaviour of the whole results from the interaction of parts. In addition, this ingredient process is required to react only to the "form" or "shape" of these mental representations, without regard to what they mean or represent—this is the substance of the claim that computation involves *formal* symbol manipulation. Thus my thought that, for example, the sun will soon A7
set, would be taken to emerge from an interaction in my mind between an ingredient process and the shape or "spelling" of various internal structures representing my knowledge that the sun does regularly set each evening, that it is currently tea time, and so forth.

The knowledge representation hypothesis may be summarised as follows:

> **Knowledge Representation Hypothesis:** Any mechanically embodied intelligent process will be comprised of structural ingredients that (a) we as external observ-

> ers 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.

Thus for example if we felt disposed to say that some process knew that dinosaurs were warm-blooded, then we would find (according, presumably, to the best explanation of how that process worked) that a certain computational ingredient in that process was understood as *representing* the (propositional) fact that dinosaurs were warm-blooded, and furthermore, that this very ingredient played a role, independent of our understanding of it as representational, in leading the process to behave in whatever way inspired us to say that it knew that fact. Presumably we would be convinced by the manner in which the process answered certain questions about their likely habitat, by assumptions it made about other aspects of their existence, by postures it adopted on suggestions as to why they may have become extinct, etc.

A careful analysis will show that. to the extent that we can make sense of it, this view that *knowing is representational* is far less evident—and perhaps, therefore, far more interesting—than is commonly believed. To do it justice requires considerable care: accounts in cognitive psychology and the philosophy of mind tend to founder on simplistic models of computation. and artificial intelligence treatments often lack the theoretical rigour necessary to bring the essence of the idea into plain view. Nonetheless, conclusion or hypothesis, it permeates current theories of mind, and has in particular led researchers in artificial intelligence to propose a spate of computational languages and calculi designed to underwrite such representation. The common goal is of course not so much to speculate on what is actually represented in any particular situation as to

uncover the general and categorical form of such representation. Thus no one would suggest how anyone actually represents facts about tea and sunsets: rather, they might posit the general form in which such beliefs would be "written" (along with other beliefs, such as that Lhasa is in Tibet, and that π is an irrational number). Constraining all plausible suggestions, however, is the requirement that they must be able to demonstrate how a particular thought could emerge from such representations—this is a crucial meta-theoretic characteristic of artificial intelligence research. It is traditionally considered insufficient merely to propose true theories that do not enable some causally effective mechanical embodiment. The standard against which such theories must ultimately judged, in other words, is whether they will serve to underwrite the construction of demonstrable, behaving artefacts. Under this general rubric knowledge representation efforts differ markedly in scope, in approach, and in detail; they differ on such crucial questions as whether or not the mental structure are modality specific (one for visual memory, another for verbal, for example). In spite of such differences, however, they manifest the shared hope that an attainable first step towards a full theory of mind will be the discovery of something like the structure of the "mechanical mentalese" in which our beliefs are inscribed.

It is natural to ask whether the knowledge representation hypothesis deserves our endorsement, but this is not the place to pursue that difficult question. Before it can fairly be asked, we would have to distinguish a strong version claiming that knowing is *necessarily* representational from a weaker version claiming merely that it is *possible* to build a representational knower. We would run straight into all the much-discussed but virtually intractable questions about what would be required to convince us that an artificially constructed process exhibited intelligent behaviour. We would certainly need a

definition of the word 'represent,' about which we will subsequently have a good deal to say. Given the current (minimal) state of our understanding, I myself see no reason to subscribe to the strong view, and remain sceptical of the weak version as well. But one of the most difficult questions is merely to ascertain what the hypothesis is actually saying—thus my interest in representation is more a concern to *make it clear* than it is to defend or deny it The entire present investigation, therefore, will be pursued under this hypothesis, not because we grant it our allegiance, but merely because it deserves our attention. A8

Given the representation hypothesis, the suggestion as to how to build self-reflective systems—a suggestion we will call the *Reflection Hypothesis*—can be summarised as follows:

> **Reflection Hypothesis:** 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 could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.

Thus the task of building a computationally reflective system is thought to reduce to, or at any rate to include, the task of providing a system with formal representations of its own constitution and behaviour. Hence a system able to imagine a world where unicorns have wings would have to construct formal representations of that fact; a system considering the adoption of a hypothesis-and-test style of investigation would have to construct formal structures representing such an inference regime.

Whatever its merit, there is ample evidence that researchers arc taken with this view. Systems such as Weyhrauch's FOL, Doyle's TMS, McCarthy's ADVICE-TAKER, Hayes' GOLUM,

and Davis' TERESIUS arc particularly explicit exemplars of just such an approach.[2] In Weyhrauch's system, for example, sentences in first-order logic arc constructed that axiomatize the behaviour of the Lisp procedures used in the course of A9 the computation (FOL is a prime example of the dual-calculus approach mentioned earlier). In Doyle's systems, explicit representations of the dependencies between beliefs and of the "reasons" the system accepts a conclusion play a causal role in the inferential process. Similar remarks hold for the other projects mentioned, as well as for a variety of other current research. In addition, it turns out on scrutiny that a great deal of current computational practice can be seen as dealing, in one way or another, with reflective abilities, particularly as exemplified by computational structures representing other computational structures. We constantly encounter examples: the wide-spread use of macros in Lisp, the use of meta-level structures in representation languages, the use of explicit non-monotonic inference rules, the popularity of meta-level rules in planning systems.[3] Such a list can be extended indefinitely; in a recent symposium Brachman reported that the love affair with *"meta-level reasoning"* was the most important theme of knowledge representation research in the last decade.[4]

4a The Relationship Between Reflection & Representation

The manner in which this discussion has been presented so far would seem to imply that the interest in *reflection* and the adoption of a *representational* stance are theoretically in-

[2]Weyhrauch (1978), Doyle (1979), McCarthy (1968), Hayes (1979), and Davis (1980a), respectively.

[3]For a discussion of macros see the various sources on Lisp mentioned in note 16 of chapter 1; meta-level rules in representation were discussed in Brachman and Smith (1980); for a collection of papers on non-monotonic reasoning see Bobrow (1980); macros are discussed in Pitman (1980).

[4]Brachman (1980).

dependent positions. I have argued in this way for a reason: to make clear that the two subjects are not the same. There is no *a priori* reason to believe that even a fully representational system should in any way be reflective or able to make anything approximating a reference to itself; similarly, there is no *proof* that a powerfully self-referential system need be constructed of representations. However—and this is the crux of the matter—the reason to raise both issues together is that they are surely, in some sense, related. If nothing else, the word 'representation' comes from 're' plus 'present', and the ability to *re-present* a world to itself is undeniably a crucial, if not *the* crucial, ingredient in reflective thought. If I reflect on my childhood, I re-present to myself my school and the rooms of my house; if I reflect on what I will do tomorrow, I bring into the view of my mind's eye the self I imagine that tomorrow I will be. If we take "representation" to describe an *ability* rather than a *structure,* reflection surely involves representation (although—and this should be kept clearly in mind—the "representation" of the knowledge representation hypothesis refers to ingredient structures, not to an activity).

It is helpful to look at the historical association between these ideas, as well to search for commonalities in content. In the early days of artificial intelligence, a search for the general patterns of intelligent reasoning led to the development of such general systems as Newell and Simon's GPS, predicate logic theorem provers, and so forth.[5] The descriptions of the subject domains were minimal but were nonetheless primarily declarative, particularly in the case of the systems based on logic. However it proved difficult to make such general systems effective in particular cases: so much of the "expertise" involved in problem solving seems domain and task specific. In reaction against such generality, therefore, a *procedural* approach emerged in which the primary focus was on the manipulation and reasoning about specific problems in simple

[5]Newell and Simon (1963); Newell and Simon (1956).

worlds.[6] Though the procedural approach in many ways solved the problem of undirected inferential meandering, it too had problems: it proved difficult to endow systems with much generality or modularity when they were simply constituted of procedures designed to manifest certain particular skills. In reaction to such brittle and parochial behaviour, researchers turned instead to the development of processes designed to work over general representations of the objects and categories of the world in which the process was designed to be embedded. Thus the *representation hypothesis* emerged in the attempt to endow systems with generality, modularity, flexibility, and so forth with respect to the embedding world, but to retain a procedural effectiveness in the control component.[7] In other words, in terms of our main discussion, representation as a method emerged as a solution to the problem of providing general and flexible ways of reflecting (not self-referentially) about the world.

Systems based on the representational approach—and it is fair to say that most of the current "expert systems" are in this tradition—have been relatively successful in certain respects, but a major lingering problem has been a narrowness and inflexibility regarding the style of reasoning these systems employ in using these representational structures. This inflexibility in *reasoning* is strikingly parallel to the inflexibility in *knowledge* that led to the first round of representational systems; researchers have therefore suggested that we need reflective systems able to deal with their own constitutions as well as with the worlds they inhabit. In other words, since the *style* of the problem is so parallel to that just sketched, it has seemed that another application of the same medicine might be appropriate. If we could inscribe general knowledge about

[6] The proceduralist view was represented particularly by a spate of dissertations emerging from MIT at the beginning of the 1970s; see for example Winograd (1972), Hewitt (1972), Sussman et al. (1971), etc.

[7] See Minsky (1975), Winograd (1975), and all of the systems reported in Brachman and Smith (1980).

how to reason in a variety of circumstances in the "mentalese" of these systems, it might be possible to design a relatively simpler inferential regime over this "meta-knowledge about reasoning," thereby engendering a flexibility and modularity regarding reasoning, just as the first representational work engendered a flexibility and modularity about the process's embedding world.

There are problems, however, in too quick an association between the two ideas, not the least of which is the question of to *whom* these various forms of re-presentation are being directed. In the normal case—that is to say, in the typical computational process built under the aegis of the knowledge representation hypothesis—a process is constituted from symbols that we as external theorists take to be representational structures; they are visible *only to the ingredient interpretive process [that is just part] of the whole*, and they are visible to that constituent process *only formally* (this is the basic claim of computation). Thus the interpreter can see them, though it is blind to the fact of their being representations. (In fact it is almost a great joke that the blindly formal ingredient process should be called an *interpreter*: when the Lisp interpreter evaluates the expression '(+ 2 3)' and returns the result '6', the last thing it knows is that the numeral '2' denotes the number *two*.) A10

Whatever is the case with the ingredient process, there is no reason to suppose that the representational structures are visible to the whole constituted process *at all,* formally or informally. That process is made out of them; there is no more *a priori* reason to suppose that they are accessible to its inspection than to suppose that a camera could take a picture of its own shutter—no more reason to suppose it is even a coherent possibility than to say that France is near Marseilles. Current practice should overwhelmingly convince us of this point: what is as tacit—what is as thoroughly lacking in self-knowledge—as the typical modern computer system? A11

The point of the argument here is not to prove that one *cannot* make such structures accessible—that one *cannot* make a representational reflective system—but to make clear that two ideas are involved. Furthermore, they are different in kind: one (representation) is a possibly powerful *method* for the construction of systems; the other (reflection) is a kind of *behaviour* we are asking our systems to exhibit. It remains a question whether the representational method will prove useful in the pursuit of the goal of reflective behaviour.

That, in a nutshell, is our overall project.

4b The Theoretical Backdrop

It takes only a moment's consideration of such questions as the relationship between representation and reflection to recognise that the current state of our understanding of such subjects is terribly inadequate. In spite of the general excitement about reflection, self-reference, and computational representation, no one has presented an underlying theory of any of these issues. The reason is simple: we are so lacking in adequate theories of the surrounding territory that, without considerable preliminary work, cogent definitions cannot even be attempted. Consider for example the case regarding self-referential reflection, where just a few examples will make this clear.

1. From the fact that a reflective system A is implemented in system B, it docs not follow that system B is thereby rendered reflective (for example, in this dissertation I will present a partially-reflective dialect of Lisp that I have implemented on a Digital Systems Corporation PDP-10, but the PDP-10 is not itself reflective). Hence A12
even a *definition* of reflection will have to be backed by theoretical apparatus capable of distinguishing between one abstract machine and another in which the first is implemented—something we are not yet able to do.

2. The notion seems to require of a computational process, and (if we subscribe to the representational hypothesis) of its interpreter, that in reflecting it "back off" one level of reference, and we lack theories both of interpreters in general, and of computational reference in particular.
3. Theories of computational interpretation will be required to clarify the confusion mentioned above regarding the relationship between reflection and representation: for a system to reflect it must re-present *for itself* its mental states; it is not sufficient for it to comprise a set of formal representations inspected *by its interpreter.* This is a distinction we encounter again and again; a failure to make it is the most common error in discussions of the plausibility of artificial intelligence from those outside the computational community, derailing the arguments of such thinkers as Searle and Fodor.[8]
4. Theories of reference will be required in order to make sense of the question of what a computational process is "thinking" about at all, whether reflective or not (for example, it may be easy to claim that when a program is manipulating data structures representing women's votes that the process as a whole is "thinking about suffrage," but what is the process thinking about when the interpreter is expanding a macro definition?).
5. Finally, if the search for reflection is taken up too enthusiastically, one is in danger of interpreting everything as evidence of reflective thinking, since what may not be reflective *explicitly* can usually be treated as *implicitly* reflective (especially given a little imagination on the part of the theorist). However we lack general guidelines on how to distinguish explicit from implicit aspects of computational structures.

[8]Searle (1980), Fodor (1978 and 1980).

Nor is our grasp of the representational question any clearer; a serious difficulty, especially since the representational endeavour has received much more attention than has reflection. Evidence of this lack can be seen in the fact that, in spite of an approximate consensus regarding the general form of the task, and substantial effort on its behalf, no representation scheme yet proposed has won substantial acceptance in the field. Again this is due at least in part to the simple absence of adequate theoretical foundations in terms of which to formulate either enterprise or solution. We do not have theories of either representation or computation in terms of which to define the terms of art currently employed in their pursuit *(representation, implementation, interpretation, control structure. data structure, inheritance,* and so forth), and are consequently without any well-specified account of what it would be to succeed, let alone of what to investigate, or of how to proceed. Numerous related theories have been developed (model theories for logic, theories of semantics for programming languages, and so forth), but they do not address the issues of knowledge representation directly, and it is surprisingly difficult to weave their various insights into a single coherent whole.

The representational consensus alluded to above, in other words, is widespread but vague; disagreements emerge on every conceivable technical point, as was demonstrated in a recent survey of the field.[9] To begin with, the central notion of "representation" remains notoriously unspecified: in spite of the intuitions mentioned above, there is remarkably little agreement on whether a representation must "re-present" in any constrained way (like an image or copy), or whether the word is synonymous with such general terms as "sign" or "symbol." A further confusion is shown by an inconsistency in usage as to what representation is a relationship between. The sub-discipline is known as the *representation of knowledge,* but in the survey just mentioned by far the majority of the respon-

[9]Brachman and Smith (1980).

dents (to the surprise of this author) claimed to use the word, albeit in a wide variety of ways, as between formal symbols *and the world about which the process is designed to reason*. Thus a KLONE structure might be said to *represent Don Quixote tilting at a windmill*; it would not be taken as representing *the fact or proposition of this activity*. In other words the majority opinion is not that we are *representing knowledge* at all, but rather, as we put it above, that *knowing is representational*.[10]

In addition, we have only a dim understanding of the relationship that holds between the purported representational structures and the ingredient process that interprets them. This relates to the crucial distinction between that interpreting process and the whole process of which it is an ingredient (whereas it is *I* who thinks of sunsets, it is at best a *constituent of my mind* that inspects a mental representation). Furthermore, there are terminological confusions: the word 'semantics' is applied to a variety of concerns, ranging from how natural language is translated into the representational structures, to what those structures represent, to how they impinge on the rational policies of the "mind" of which they are a part, to what functions are computed by the interpreting process, etc. The term 'interpretation' (to take another example) has two relatively well-specified but quite independent meanings, one of computational origin, the other more philosophical; how the two relate remains so far unexplicated, although, as was just mentioned, they are strikingly distinct.

A13

A14

Unfortunately, such general terminological problems are just the tip of an iceberg. When we consider our specific representational proposals, we are faced with a plethora of apparently incomparable technical words and phrases. *Node*, *frame*, *unit*, *concept*, *schema*, *script*, *pattern*, *class*, and *plan*, for example, are all popular terms with similar connotations and ill-defined meaning.[11] The theoretical situation (this may

[10]See the introduction to Brachman and Smith (1980).

[11]References on *node*, *frame*, *unit*, *concept*, *schema*, *script*, *pattern*, *class*,

not be so harmful in terms of more practical goals) is further hindered by the tendency for representational research to be reported in a rather demonstrative fashion: researchers typically exhibit particular formal systems that (often quite impressively) embody their insights, but that are defined using formal terms peculiar to the system at hand. We are left on our own to induce the relevant generalities and to locate them in our evolving conception of the representation enterprise as a whole. Furthermore, such practice makes comparison and discussion of technical details always problematic and often impossible, defeating attempts to build on previous work.

This lack of grounding and focus has not passed unnoticed: in various quarters one hears the suggestion that, unless severely constrained, the entire representation enterprise may be ill-conceived—that we should turn instead to considerations of particular epistemological issues (such as how we reason about, say, liquids or actions), and should use as our technical base the traditional formal systems (logic, Lisp, and so forth) that representation schemes were originally designed to replace.[12] In defense of this view two kinds of argument are often advanced. The first is that questions about the *central* cognitive faculty are at the very least premature, and more seriously may for principled reasons never succumb to the kind of rigorous scientific analysis that characterizes recent studies of the *peripheral* aspects of mind: vision, audition, grammar, manipulation, and so forth.[13] The other argument is that logic as developed by the logicians is in itself sufficient; that all we need is a set of ideas about what axioms and inference

. . .

and *plan* can be found in the various references provided in Brachman and Smith (1980).

[12]See in particular Hayes (1978).

[13]The distinction between central and peripheral aspects of mind is articulated in Nilsson (1981); on the impossibility of central AI (Nilsson himself feels that the central faculty will quite definitely succumb to AI's techniques) see Dreyfus (1972) and Fodor (1980 and forthcoming).

protocols are best to adopt.[14] But such doubts cannot be said to have deterred the whole of the community: the survey just mentioned lists more than thirty new representation systems under active development.

The strength of this persistence is worth noting, especially in connection with the theoretical difficulties just sketched. There can be no doubt that there are scores of difficult problems: we have just barely touched on some of the most striking. But it would be a mistake to conclude in discouragement that the *enterprise* is doomed, or to retreat to the meta-theoretic stability of adjacent fields (like proof theory, model theory, programming language semantics, and so forth). The moral is at once more difficult and yet more hopeful. What is demanded is that we stay true to these undeniably powerful ideas, and attempt to develop adequate theoretical structures on this home ground. It is true that any satisfactory theory of computational reflection must ultimately rest, more or less explicitly, on theories of computation, of intensionality, of objectification, of semantics and reference, of implicitness, of formality, of computation, of interpretation, of representation, and so forth. On the other hand as a community we have a great deal of practice that often embodies intuitions that we are unable to formulate coherently. The wealth of programs and systems we have built often betray—sometimes in surprising ways—patterns and insights that eluded our conscious thoughts in the course of their development. What is mandated is a *rational reconstruction* of those intuitions and of that practice.

In the case of designing reflective systems, such a reconstruction is curiously urgent. In fact this long introductory story ends with an odd twist—one that "ups the ante" in the search for a carefully formulated theory, and suggests that practical progress will be impeded until we take up the theoretical task. In general, it is of course possible (some would

[14]Nilsson (1981).

even advocate this approach) to build an instance of a class of artefact before formulating a theory of it. The era of sail boats, it has often been pointed out, was already drawing to a close just as the theory of airfoils and lift was being formulated—the [very] theory that, at least at the present time, best explains how those sailboats worked. However there are a number of reasons why such an approach may be ruled out in the present case. For one thing, in constructing a reflective calculus one must support arbitrary levels of meta-knowledge and self-modelling, and it is self-evident that confusion and complexity will multiply unchecked when one adds such facilities to an only partially understood formalism. It is simply likely to be unmanageably complicated to *attempt* to build a self-referential system unaided by the clarifying structure of a prior theory. The complexities surrounding the use of APPLY in Lisp (and the caution with which it has consequently come to be treated) bear witness to this fact. However there is a more serious problem. If one subscribes to the knowledge representation hypothesis, it becomes an integral part of developing self-descriptive systems to provide, encoded within the representational medium, an account of (roughly) the syntax, semantics, and reasoning behaviour of that formalism. In other words, if we are to build a process that "knows" about itself: and *if we subscribe to the view that knowing is representational,* then we are committed to providing that system with a *representation* of the self-knowledge with which we aim to endow it. That is, we must have an adequate theories of computational representation and reflection *explicitly formulated,* since *an encoding of that theory is mandated to play a causal role as an actual ingredient in the reflective device*.

Knowledge of any sort—and self-knowledge is no exception—is always theory relative. The representation hypothesis implies that our theories of reasoning and reflection must be explicit. We have argued that this is a substantial, if widely ac-

cepted, hypothesis. One reason to find it plausible comes from viewing the entire enterprise as an attempt to communicate our thought patterns and cognitive styles—including our reflective abilities—to these emergent machines. It may at some point be possible for understanding to be tacitly communicated between humans and system they have constructed. In the meantime, however, while we humans might make do with a rich but unarticulated understanding of computation, representation, and reflection, we must not forget that computers do not [yet] share with us our tacit understanding of what they are. A14

Annotations[1]

A1 5/-1:6/1 A brief discussion of Mantiq (including a note on the provenance of the name) is provided in §1 of the Cover to this chapters 3a–3c.

It is not clear to me now (2012), however, from notes that were written about Mantiq at the time, that the description in the text is entirely accurate. In particular, the phrase "in which what happened took effect in virtue of declarative descriptions of what was to happen" (5/-1/-2:6/0/1) conveys the untenable suggestion that all Mantiq structures would have only declarative force. As is clear from philosophy of language—to say nothing of the Carroll paradoxes—statement alone is not by itself sufficient to engender action. In addition, as in 3Lisp, the aim for Mantiq was to have events, structures, and phenomena co-exist on an equivalent ontological plane (though at a different semantic level!) with descriptions of those self-same events, structures, and phenomena—i.e., without either having ontic priority (a bit of a conceit).

It is true, though, that the emphasis in Mantiq was to be on *description*—something for which I still believe support (and theory) to be woefully missing in computational languages. For numerous reasons (cf. «where?») I do not believe RDF, XML, OWL, etc., come close to filling the bill.

A2 6/1/4 To minimise confusion I explicitly flag chapter references that refer to chapters in the dissertation, of which only chapter 1 is included in this Volume,[5] so as to distinguish them from references to chapters in the present volume.

A3 6/1/-6:-3 This project of developing an architecture in which structural identity[2] could serve as a proxy for intensional identity was one of Mantiq's primary design aims. It was also something on which I had spent a lot of time working, before 3Lisp was designed. The basic idea was to define a rather abstract conception of a structural field (rather like an abstract memory), implemented by a background by concurrent relaxation algorithms, so that structural identity (of the sort that would be tested by an analogue of Lisp's EQ) would mimic identity of meaning on a plausible if necessarily relatively fine-grained way. For example: internal analogues of such expressions as

1. References are in the form *page*/*paragraph*/*line*; with ranges (of any type) indicated as *x*:*y*. For details see the explanation on p. •

2. The text says 'syntactic' identity—but it is very clear that I meant the directly-inspectible identity of internal computational structures.

(λy . y+3) and (λx . 3+x) would be token (not just type) identical.

There are legions of issues lurking behind this suggestion—including, for example, concerns of the sort articulated in "The Correspondence Continuum" (ch. 11), that different granulaties of intensional identity are appropriate in different circumstances. Part of what I had explored, in the emerging Mantiq design, was the exploitation of reflection to obtain more or less fine-grained access to these sorts of structural granularity, in line with the overall philosophy of providing contextually sensitive ways of making things more or less explicit (so that operator order, for example, or the forms of equivation addressed in de Morgan's laws, could be "seen" or "not seen," depending on purpose and perspective, in a flexible way).

A4 6/-1/-3:-2 In the end I did define a reflective version of the λ-calculus, in an attempt to communicate to Jon Barwise why I thought reflection was interesting, and how it worked. See §6 of the Introduction and annotation A41 in ch. 4.

A5 7/0/-3:-1 As indicated here, the aim of developing 3Lisp was to work out a semantical framework that integrated an understanding of reference and description into an account of computational activity. It was because of this motivating purpose that I felt that the "delivery" of 3Lisp failed, since even though the notion of reflection was positively received, the semantical framework on which it was based was ignored. Cf. the discussion in the Introduction, especially at ..... in §1, and in §6, where among other things I suggest that this failure stemmed from untenable ontological as well as semantical presuppositions underlying the understanding of reference that I employed in 3Lisp's design, and that still remains our default theoretical approach to these subjects.

A6 7/-1/-5:-4 It is the claim that writing programs requires that one make one's ideas *completely* explicit with which I was disagreeing. I certainly believed (and still do today) that constructing a program requires a kind of explicitisation that is extraordinarily demanding—far more so than those who have not programmed are ever likely to realise.

A7 11/1/-8:-6 Though endorsing the formality condition here, my belief in its truth (of real-world computation) had already begun to erode. By the time that "Reflection and Semantics in Lisp" (ch. 4) was published in 1983, I was close to be willing to deny its truth (though I continue

to believe that it is based on a profoundly deep insight). See annotation A15 in ch. 4, and Volume II of AOS.

A8 14/0/4 'Sceptical' is the operative word. At the time I was neither prepared to endorse or to deny the representational view, in spite of its ubiquitous allegiance in Artificial Intelligence at the time (Haugeland's GOFAI did not come in for resounding critique until later in the decade). See "Registration and Registration" in *Indiscrete Affairs*, Vol. II.

A9 15/0/4 For discussion of the notion of a *dual calculus* see •1/-1/-5:-2 and •21/0:1 in ch. 3b.

A10 18/1 I was less clear on these issues here than I should have been. Cf. the discussions in «ref where ingredient and specificational views are talked about, etc.; certainly including ch. 4 and 100 Billion Lines». In addition, there are issues about the relation between personal and sub-personal levels (cf. annotation A26 of ch. 3b, p •115) which I later came to recognise as profoundly important, but to which in 1981 I was not appropriately aware.

A11 18/-1/-2:-1 In 1980, the year before the dissertation was written, I had presented a response to Charles Taylor's paper "Cognitive Psychology."[3] One of Taylor's points was that human life is drenched in a background of inexorable implicitness, which he took to be antithetical to (what he took to be) the intrinsic explicitness of computation. While taking no exception to his account of human life, I disagreed with his claim that computation is anything like as explicit as he suggested. Ever since I had started programming in the late 1960s, I had had (and still have) a deep sense that there was far more that is tacit and implicit in the computational realm than is commonly recognised in the reigning mythos.

A12 19/-15:6 The PDP-10, a mainframe computer built by the Digital Equipment Corporation (DEC), formed the backbone of the computational resources at the artificial intelligence laboratories at MIT, Stanford, and Carnegie-Mellon. Among other things, it was the computer on which time-sharing was initially developed; it was also the machine on which Lisp ran, in which all early AI projects were implemented. As a result, it loomed large in the imagination of early AI theorists.

A13 22/1/6:7 This passage, too (as well as 18/1; see A10, above), would have been

3. The response, entitled "The Significance of Computational Psychology," was presented on March 25, 1980 at a Conference on Artificial Intelligence and Philosophy, held at the Center for Advanced Studies in the Behavioral Sciences in Stanford, California.

better if framed in terms of a personal/subpersonal distinction. Cf. annotation A26 of ch. 3b, p. •115.

A14 22/1/8:13 «Refer to general discussions in Intro? Also point forward to 100 Billion»

A15 The word 'yet,' present in drafts written right up until the point of-submission, was for unknown reasons deleted in the submitted version.

Procedural Relection in Programming Languages

3b Introduction

The successful development of a general reflective calculus based on the knowledge representation hypothesis will depend on the prior solution of three problems:

1. The provision of a computationally tractable and epistemologically adequate descriptive language;
2. The formulation of a unified theory of computation and representation; and
3. The demonstration of how a computational system can reason effectively and consequentially about its own inference processes.

The first of these issues is the collective goal of present knowledge representation research; though much studied, it has met with only partial success. The problems involved are enormous, covering such diverse issues as adequate theories of intensionality, methods of indexing and grouping representation A1
al structures, and support for variations in assertional force. In spite of its centrality, however, it will not be pursued here, in part because it is so ill-constrained. The second, though it is occasionally acknowledged to be important, is a much less well publicised issue, having received (so far as I know) almost no direct attention. As a consequence, every representation sys- A2
tem proposed to date exemplifies what I will call a **dual-calculus** approach: a procedural calculus (usually Lisp) is conjoined with a declarative formalism (an encoding of predicate logic, frames, etc.). Even such purportedly unified systems as Prolog[1] can be shown to manifest this dual-calculus structure. I will A3

1. Prolog has been presented in a variety of papers; see for example Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977). The conception of logic as a programming language (with which I radically disagree) is presented in Kowalski (1974 and 1979).

in passing suggest that this dual-calculus style is unnecessary and indicative of serious shortcomings in our conception of the representational endeavour. However this issue too will be largely ignored.

In this dissertation my focus instead will be on the third problem: the question of making the inferential or interpretive aspects of a computational process themselves accessible as a valid domain of reasoning. I will show how to construct a computational system whose active interpretation is controlled by structures themselves available for inspection, modification, and manipulation, in ways that allow a process to shift smoothly between dealing with a given subject or task domain, and dealing with its own reasoning processes over that domain. In computational terms, the question is one of how to construct a program able to reason about and affect its own interpretation—i.e., of how to define a calculus with a reflectively accessible control structure. A4

1a General Overview

The term "reflection" does not name a previously well-defined question to which I propose a particular solution (although logic's *reflection principles* are not unrelated). Before I can present a theory of what reflection comes to, and how it can be demonstrated, therefore, I will have to give an account of what reflection is. In the next section, by way of introduction, I will identify six characteristics that I take to distinguish all reflective behaviour. Then, since I will be primarily concerned with **computational reflection**, I will sketch the model of computation on which the analysis will be based, and will set the general approach to reflection to be adopted into a computational context. In addition, once a working vocabulary of computational concepts has been set out, I will be able to define what I will mean by **procedural reflection**—an even smaller and more circumscribed notion than computational reflection in

general. All of these preliminaries are necessary in order to enable the formulation of an attainable set of goals.

Thus prepared, I will set forth on the analysis itself. As a technical device, over the course of the dissertation I will develop three successive dialects of Lisp to serve as illustrations, and to provide a technical ground in which to work out in detail the theory of reflection to be proposed. I should say at the outset, however, that this focus on Lisp should not mislead the reader into thinking that the basic reflective architecture I propose—or the principles endorsed in its design—are in any important sense Lisp specific. Lisp was chosen because it is simple, powerful, and uniquely suited for reflection in two ways: it already embodies protocols whereby programs are represented in first-class accessible (data) structures, and it is a convenient formalism in which to express its own meta-theory—especially given that I will use a variant of the λ-calculus as a mathematical meta-language (this convenience holds especially in a statically scoped dialect of the sort that I will ultimately adopt). Nevertheless, as I will discuss in the concluding chapter [of the dissertation], it would be possible to construct A5 a reflective dialect of Fortran, Smalltalk, or any other procedural calculus, by pursuing essentially the same approach as I will demonstrate here for Lisp.

The first Lisp dialect (called **1Lisp**) will be an example intended to summarise current practice, primarily for comparison and pedagogical purposes. The second (**2Lisp**) differs rather substantially from 1Lisp, in that it is modified with reference to a theory of declarative denotational semantics (i.e., a theory of the denotational significance of s-expressions) formulated *independent of the behaviour of* (what computer science calls) *the "interpreter."* The interpreter is then subsequently defined with respect to this theory of attributed se- A6 mantics, so that the result of processing of an expression—i.e., the value of the function computed by the basic interpretation

process—is a *normal-form co-designator* of the input expression. I will call 2Lisp a **semantically rationalised** dialect, and will argue that it makes explicit much of the understanding of Lisp that tacitly organises most programmers' understanding of Lisp but that has never been made an articulated part of Lisp theory. Finally, a procedurally reflective Lisp called **3Lisp** will be developed, semantically and structurally based on 2Lisp, but modified so that reflective procedures are supported, as a vehicle with which to engender the sorts of procedural reflection we will by then have set as our goal. 3Lisp differs from 2Lisp in a variety of ways, of which the most important is the provision, at any point in the course of the computation, for a program to *reflect* and thereby obtain fully articulated "descriptions," formulated with respect to a primitively endorsed and encoded theory, of the state of the interpretation process that was in effect at the moment of reflection. In this particular case, this will mean that a 3Lisp program will be able to access, inspect, and modify standard 3Lisp normal-form designators of both the environment and continuation structures that were in effect a moment before.

A7

More specifically, 1Lisp, like Lisp 1.5 and all Lisp dialects in current use, is at heart a *first-order* language, employing meta-syntactic facilities and dynamic variable scoping protocols to partially mimic higher-order functionality. Because of its metasyntactic powers (paradigmatically exemplified by the primitive QUOTE), 1Lisp contains a variety of inchoate reflective features, all of which I will examine in some detail: support for metacircular interpreters, explicit names for the primitive processor functions (EVAL and APPLY), the ability to *mention* program fragments, protocols for expanding macros, and so on and so forth. Though I will ultimately criticise much of 1Lisp's structure (and its underlying theory), I will document its properties in part to serve as a contrast for the subsequent dialects, and in part because, being familiar, 1Lisp can serve as a base in which to ground the analysis.

A8

After introducing 1Lisp, but before attempting to construct a reflective dialect, I will subject 1Lisp to rather thorough semantical scrutiny. This project, and the reconstruction that results, will occupy well over half the dissertation. The reason is that the analysis will require a reconstruction not only of Lisp but of computational semantics in general. I will argue in particular that it is crucial, in order to develop a comprehensible reflective calculus, to have a semantical analysis of that calculus that makes explicit the tacit attribution of significance that I will claim characterises every computational system. I take this attribution of semantical import to computational expressions to be *prior* to any account of what *happens* to those expressions: thus I will argue for an analysis of computational formulae in which **declarative import** and **procedural consequence** are independently formulated. I claim, in other words, that programming languages are better understood in terms of *two* semantical treatments—one declarative, one procedural—rather than in terms of a single one, as is exemplified by current approaches (although interactions between them may require that these two semantical accounts be formulated in conjunction).

A9 A10 A11

This semantical reconstruction is at heart a comparison and combination of the standard semantics of programming languages on the one hand, and the semantics of natural human languages and of descriptive and declarative languages such as predicate logic, the λ-calculus, and mathematics, on the other. Neither will survive intact: the approach I will ultimately adopt is not strictly compositional in the standard sense (although it is recursively specifiable), nor are the declarative and procedural facets entirely separate. In particular, the procedural consequence of executing a given expression may affect the subsequent context of use that determines what another expression declaratively designates. Nor are the consequences of this approach minor. For example, I will show that

A12

the traditional notion of *evaluation*, in terms of which all Lisps to date have been defined, is both confusing and confused, and must be separated into independent notions of **reference** and **simplification**. I will be able to show, in particular, that ILisp "evaluator" de-references some expressions (such meta-syntactic terms as (QUOTE X), for example), and does not dereference others (such as the numerals and T and NIL). I will argue instead for what I will call a **semantically rationalised** dialect, in which the simplification and reference primitives are kept strictly distinct.

The basic thesis on which this work depends is that semantical cleanliness (along the lines suggested above) is by far the most important pre-requisite to any coherent treatment of reflection. However, as well as advocating *semantically rationalised* computational calculi, in the Lisp case I will also espouse an aesthetic I call **category alignment**, by which I mean that there should be a strict category-category correspondence across the four major axes in terms of which a computation calculus is analysed: (i) notation; (ii) abstract structure; (iii) A13
declarative semantics; and (iv) procedural consequence (a mandate satisfied by no extant Lisp dialect). In particular, in the dialects I design and present here, I will insist: that each *notational* class be parsed into a distinct structural class; that each structural class be treated in a uniform way by the primitive processor; that each structural class serve as the normal-form designator of each semantic class; and so forth.

Category alignment is an aesthetic with consequence. I will show that the ILisp programmer (i.e., all existing Lisp programmers) must in certain situations resort to meta-syntactic machinery merely because ILisp fails to satisfy this mild requirement (in particular, ILisp *lists*, which are themselves a derivative class formed from some pairs and one atom, serve semantically to encode both function applications and enumerations). Though it by no means has the same status as se-

mantical hygiene, categorical elegance will also prove almost indispensable, especially from a practical point of view, in the drive towards reflection. A14

Once these theoretical positions have been formulated, I will be in a position to design 2Lisp. Like Scheme and the λ-calculus, 2Lisp is a higher-order formalism: consequently, it is statically scoped, and treats the function position of an application as a standard extensional position. 2Lisp is of course formulated in terms of the rationalised semantics, according to which declarative semantics must be formulated for all expressions prior to, and independent of, the specification of how they are treated by the primitive processor. Consequently, and unlike Scheme, the 2Lisp processor is based on a regimen of **normalisation**, according to which each expression is taken into a normal-form designator of its referent, where the notion of *normal-form* is defined in part with reference to the semantic type of the symbol's *designation*, rather than (as in the case of the λ-calculus) in terms of the further non-applicability of a set of syntactic reduction rules. A15

2Lisp's normal-form designators are environment independent and side-effect free; thus the concept of a *closure* can be reconstructed as a *normal-form function designator*. Since normalisation is a form of simplification, and is therefore *designation-preserving,* meta-structural expressions (terms that designate other terms in the language) are not de-referenced upon normalisation, as they are when evaluated. I therefore call the 2Lisp processor **semantically flat**, since it stays at a semantically fixed level (although explicit referencing and dereferencing primitives are also provided, to facilitate explicit shifts in level of designation). A16

3Lisp is straightforwardly defined as an extension of 2Lisp, with respect to an explicitly articulated procedural theory of

3Lisp embedded in 3Lisp structures. This embedded theory, called the **reflective model**, though superficially resembling a metacircular interpreter (as shown by a glance at the code, given in figure 15 on p. ·99), is *causally connected* to the workings of the underlying calculus in critical and primitive ways. The reflective model is similar in structure to the procedural fragment of the meta-theoretic characterisation of 2Lisp that was encoded in the λ-calculus: it is this incorporation into a system of a theory of its own operations that makes 3Lisp, like any possible reflective system, inherently theory relative. For example, whereas *environments* and *continuations* will up until this point have been theoretical posits, mentioned only in the theorist's meta-language as a way of explaining Lisp's behaviour, in 3Lisp such entities move from the semantical domain of the external theoretical meta-language into the semantical domain of the object language, in such a way that environment and continuation designators emerge as part of the primitive behaviour of 3Lisp protocols. A17 A18

More specifically, arbitrary 3Lisp **reflective procedures** can bind as arguments (designators of) the continuation and environment structure of the interpreter that *would have been in effect at the moment the reflective procedure was called*, had the machine been running all along in virtue of the explicit interpretation of the prior program, mediated by the reflective model. Furthermore, by constructing and/or modifying these designators, and resuming the process below, such a reflective procedure may arbitrarily control the processing of programs at the level beneath it. Because reflection may recurse arbitrarily, 3Lisp is most simply defined in terms of the following ideal: A19

> *An infinite tower of 3Lisp processes, each engendering the process immediately below, in virtue of running a copy of the reflective model.*

Under such an account, the use of reflective procedures amounts to running simple procedures at arbitrary levels in this reflective hierarchy. Both a straightforward implementation and a conceptual analysis are provided to demonstrate that such a machine is nevertheless finite.

3Lisp's reflective levels are not unlike the levels in a typed logic or set theory, although of course each reflective level contains an omega-order untyped computational calculus essentially isomorphic to (the extensional portion of) 2Lisp. Reflective levels, in other words, are at once stronger and more encompassing than are the order levels of traditional systems. The locus of agency in each 3Lisp level, on the other hand, that distinguishes one computational level from the next, is a notion without precedent in logical or mathematical traditions.

The architecture of 3Lisp allows us to unify three concepts of traditional programming languages that are typically independent (three concepts we will have explored separately in 1Lisp):

1. The ability to support metacircular interpreters;
2. The provision of explicit names for the primitive interpretive procedures (EVAL and APPLY in standard Lisp dialects); and
3. The inclusion of procedures that access the state of the implementation (usually provided as part of a programming environment, for debugging purposes).

I will show how all such behaviours can be defined within a pure version of 3Lisp (i.e., independent of implementation), since all aspects of the state of the 3Lisp interpretation process are available, with sufficient reflection, as objectified entities within the 3Lisp structural field.

The dissertation concludes by drawing back from the details of Lisp development, in order to show how the techniques

employed in this one particular case could be used in the construction of other reflective languages—reflective dialects of current formalisms, or other new systems built from the ground up. I will show, in particular, how this approach to reflection may be integrated with notions of data abstraction and message passing—two (related) concepts commanding considerable current attention, that might seem on the surface incompatible with the notion of a system-wide declarative semantics.Fortunately, I will be able to show that this early impression is false—that procedurally reflective *and semantically rationalised* variants on these types of languages could be readily constructed as well. A20

Besides the basic results on reflection, there are a variety of other lessons to be taken from the investigation, of which the integration of declarative import and procedural consequence in a unified and rationalised semantics is undoubtedly the most important. The rejection of evaluation, in favour of separate simplification and de-referencing protocols, is the major, but not the only, consequence of this revised semantical approach. The matter of category alignment, and the constant question of the proper use of metastructural machinery, while of course not formal results, are nonetheless important permeating themes. Finally, the unification of a variety of practices that until now have be treated independently—macros, metacircular interpreters, EVAL and APPLY, quotation, implementation-dependent debugging routines, and so forth—should convince the reader of one of the dissertations most important claims: procedural reflection is not a radically new idea; tentative steps in this direction have been taken in many areas of current practice. The present contribution—fully in the traditional spirit of rational reconstruction—is merely one of making explicit what we all already knew.

✦ ✦ ✦

I conclude this brief introduction with three footnotes.

First, given the flavour of the discussion so far, the reader may be tempted to conclude that the primary emphasis of this report is on procedural, rather than on representational, concerns (an impression that will only be reinforced by a quick glance through later [dissertation] chapters). This impression is in part illusory; as I will explain at a number of points. these topics are pursued in a procedural context because it is *simpler* than attempting to do so in a poorly understood representational or descriptive system. All of the substantive issues, however, have their immediate counterparts in the declarative aspects of reflection, especially when such declarative structures are integrated into a computational framework. This investigation has been carried on with the parallel declarative issues kept firmly in mind; the attribution of a declarative semantics to Lisp s-expressions will also reveal my representational bias. As I mentioned in the preface, the decision to first explore reflection in a procedural context should be taken as methodological, rather than as substantive. Furthermore, it is towards a *unified* system that I ultimate want to aim. One of the morals underlying this reconstruction is that the boundaries between these two types of calculus should ultimately be dismantled. A21

Second. as this last comment suggests, and as the unified treatment of semantics betrays, I consider it important to unify the theoretical vocabularies of the *declarative tradition* (logic, philosophy, and to a certain extent mathematics) with the *procedural tradition* (primarily computer science). I view the semantical approach adopted here as but a first step in that direction; as suggested in the first paragraph, a fully unified treatment remains an as-yet unattained goal. Nonetheless, I have expended some effort in the work reported here to develop and present a single semantical and conceptual position A22 that draws on the insights and techniques of both of these disciplines.

Third and finally, as the very first paragraph of this chapter suggests, the dissertation is offered as the first step in a general investigation into the construction of *generally* reflective computational calculi to be based on more fully integrated theories of representation and computation. In spite of its reflective powers, and in spite of its declarative semantics, 3Lisp cannot properly be called fully reflective, since 3Lisp structures do not form a descriptive language (nor would any other procedurally reflective programming language that might be developed in the future, based on techniques set forth here, have any claim to the more general term). This is not so much because the 3Lisp structures lack expressive power (although 3Lisp has no quantificational operators, implying that even if it were viewed as a descriptive language it would remain algebraic), but rather because 3Lisp expressions are devoid of *assertional force*. There is, in brief, no way to *say anything* in such a formalism. One can set *x* to 3, in 3Lisp or any other procedural (i.e., programming) language; one can test whether *x* is 3; but one cannot say *that x* is 3. Nevertheless, I contend that the insights won on the behalf of 3Lisp will ultimately prove useful in the development of more radical, generally reflective systems.

In sum, I hope to convince the reader that, although it will be of some interest on its own, 3Lisp is only a corollary of the major theses adopted in its development.

1b The Concept of Reflection

In this section I will look more carefully at the term "reflection," both in general and in the computational case, and also specify what I would consider an acceptable theory of such a phenomenon. The structure of the solution I will eventually adopt will be presented only in §1·e, after discussing in §1·c the attendant model of computation on which it is based. and in §1·d the conception of computational semantics to be adopted. Before presenting any of that preparatory material, however, it helps to know where we are headed.

1b·i The Reflection and Representation Hypotheses

In the prologue I sketched in broad strokes some of the roles that reflection plays in general mental life. In order to focus the discussion, this section consider in more detail what I will mean by the more restricted phrase *computational reflection*. On one reading this term might refer to a successful computational model of general reflective thinking. For example, if you were able to formulate what human reflection comes to (more precisely than I have been able to do), and were then able to construct a computational model embodying or exhibiting such behaviour, you would have some reason to claim that you had demonstrated computational reflection, in the sense of a *computational process that exhibited authentic reflective activity*.

Though I have undertaken this work with this larger goal in mind, my use of the phrase is more modest, in two important ways.

First, in this dissertation I take no stand on the question of whether computational processes are able to "think" or "reason" at all, in, as it were, their own right. Certainly it would seem that most of what we take computational systems to do is *attributed*, in a way that is radically different from the situation regarding our interpretations of the actions of other people. In particular, humans are first-class bearers of what is called **semantic originality**: they themselves are able to mean, without some observer having to attribute meaning to them. Computational processes, on the other hand, are at least not yet semantically original; to the extent they can be said to mean or refer at all, they do so **derivatively**, in virtue of some human finding that a convenient description (I duck the question as to whether it is a convenient *truth* or a convenient *fiction*).[2] For example, if, as you read this, you rationally and intentionally say "*I am now reading section 1b·i*," you succeed in referring to this section, without the aid of attendant observers. You do

A23

2. For a discussion of the semantical properties of computational systems see for example Fodor (1980), Fodor (1978), and Haugeland (1978).

so *because we define the words that way*; reference and meaning and so on are not just paradigmatically but definitionally what people do. In other words your actions are the definitional locus of reference; the rest is hypothesis and falsifiable theory. If on the other hand I "inquire" of my home computer as to the address of a friend's farm. and it "tells me" that it is on the west coast of Scotland, the computer has not referred to Scotland in any full-blooded sense—it hasn't a clue as to what or where Scotland is. Rather, it has merely typed out an address that is probably stored in an ASCII code somewhere inside it, and *I* supply the reference relationship between that spelled word and the country in the British Isles.

The *reflection hypothesis* spelled out in the prologue, about how computational models of reflection might be constructed, embodied this cautionary stance: I said there that *in as much as a computational process can be constructed to reason at all,* it could be made to reason reflectively in a certain fashion. Thus I will take the topic of computational reflection to be restricted to those computational processes that, for similar purposes, *we find it convenient to describe as reasoning reflectively*. In sum, I avoid completely the question of whether the "reflectiveness" embodied in our computational models is authentically borne, A24
or derivatively ascribed.

Setting aside worries about semantic originality is one reduction in scope; I also adopt another. Again, in the prologue, I spoke of reflection as if it encompassed contemplative consideration not only of one's self but also of one's world (and one's place therein). While I will discuss the relationship between reflection and self-reference in more detail below, it is important to acknowledge that the focus of this investigation is almost entirely on the "selfish" part of reflection: on what it is to construct computational systems able to deal *with their own ingredient structures and operations* as explicit subject matters.

The reasons for this constraint are worth spelling out. The A25

restriction might seem to arise for simple reasons, such as that this is an easier and better-constrained subject matter (I certainly do not consider myself in a position to postulate models of thinking about external worlds). But in fact the restriction arises for deeper reasons, again having to do with the reflection hypothesis. In the architectures I develop, I consider only *internal* or *interior* processes, able to reflect on *interior* structures, which is the only world that those internal processes conceivably can have any access to. Lisp processors (interpreters), in particular, have no access to anything except fields of s-expressions; they do not interact with the world directly, but rather in virtue of running programs, engender more complex processes that interact with the world.

This "interior" sense of language processors interacts crucially with the reflection hypothesis, especially in conjunction with the representation hypothesis. Not only can we restrict to our attention to ingredient processes "reasoning about" (computing over. whatever) internal computational structures, we can restrict our attention to processes *that shift their (extensional) attention to meta-structural terms*. For consider: if it turns out that I am a computational system, consisting of an ingredient process P manipulating formal representations of my knowledge of the world, then according to the representation hypothesis, when I think, say, about Virginia Falls on the Nahanni River in northern Canada, my ingredient processor P is manipulating *representations* that are about Virginia Falls. Suppose. then, that I back off a step and comment to myself that whenever I should be writing another sentence I have a tendency instead to think about Virginia Falls. What do we suppose that my processor P is doing now? Presumably ("presumably", at least, according to the Knowledge Representation Hypothesis. which, it is important to reiterate, we are under no compulsion to believe) my processor P is now manipulating *representations of my representations of Virginia Falls*. In other

words, *because we are focused on the behaviour of interior processes*, not on compositionally constituted processes, our exclusive focus on self-referential aspects of those processes is all we can do (given our two governing hypotheses) to uncover the structure of constituted, genuine reflective thought. A26

The same point can be put another way. The reflection hypothesis docs not state that, in the circumstance just described, P will *reflect* on the knowledge structures representing Virginia Falls (in some weird and wondrous way)—this would be an unhappy proposal, since it would not offer any hope of an *explanation* of reflection. On pain of circularity, reflective behaviour—the subject matter to be explained—should not occur in the explanation. Rather, the reflection hypothesis is at once much stronger and more tractable (although perhaps for that very reason less plausible): it posits, as an explanation of the mechanism of reflection, that the constituent interior processes compute over a *different kind of symbol*. The most important feature of the reflection hypothesis, in other words, is its tacit assumption that the computation engendering reflective reasoning, although it may be over a different kind of structure, is nonetheless *similar in kind* to the sorts of computation that regularly proceed over normal structures. (In this way it makes good on the background project of naturalising reflection.) A27

In sum, it is methodological allegiance to the Knowledge Representation Hypothesis, rather than a limited interest in introspection, that underwrites my self-referential stance. Though I will not discuss this meta-theoretic position further, it is crucial that it be understood, for it is only because of it that I have any right to call this inquiry a study of *reflection*, rather than a (presumably less interesting) study of *computational self-reference*. A28

1b·ii Reflection in Computational Formalisms

Turn, then, to the question of what it would be to make a computational process reflective in the sense just described.

At its heart, the problem derives from the fact that in traditional computational formalisms the behaviour and state of the interpretation process are not accessible to the reasoning procedures: the interpreter forms part of the tacit background in terms of which the reasoning processes work. Plus, in the majority of programming languages, and in all representation languages, only the uninterpreted data structures lie within the reach of a program. A few languages, such as Lisp and Snobol, A29
extend this basic provision by allowing program structures to be examined, constructed, and manipulated as first class entities. What has never before been provided is a high level language in which the process that interprets those programs is also visible and subject to modification and scrutiny. Therefore such matters as whether the interpreter is using a depth-first control strategy, whether free variables are dynamically scoped, how long the current problem has been under investigation, or what caused the interpreter to start up the current procedure, remain by and large outside the realm of reference of standard representational structures. One way in which this limitation is partially overcome in some programming languages is to allow procedures access to the structures of the *implementation* (examples: MDL, InterLISP, etc.[3]), although such a solution is inelegant in the extreme, defeats portability and coherence, lacks generality, and in general exhibits a variety of misfeatures that I will examine in due course. In more representational or declarative contexts no such mechanism has been demonstrated, although a need for some sort of reflective power has appeared in a variety of contexts (such as for overriding defaults, gracefully handling contradictions, etc.).

A striking example comes up in problem-solving: the issue

3. Such facilities as are provided in MDL are described in Galley and Pfister (1975); those in InterLISP, in Teitelman (1978).

is one of enabling simple declarative statements to be made about how the deduction operation should proceed For example, it is sometimes suggested that a *default* should be implemented by a deductive regime that accepts inferences of the following non-monotonic variety (i.e., if "not P" cannot be proved, then deduce P):

$$\frac{\neg\vdash \neg P}{P} \qquad [1]$$

Though it is not difficult to build a problem solver that *embodies* some such behaviour (at least on some computable reading of "not provable"), one typically does not want such a rule to be obeyed indiscriminately, independent of context or domain. There are, in other words, usually constraints on when such inferences are appropriate—having to do with, say, how crucially the problem needs a reliable answer, or with whether other less heuristic approaches have been tried first. What people writing problem-solver systems have wanted is a way to write down specific instances of something like [1] that explicitly refer both to the subject domain *and to the state of the deductive apparatus*, which, *in virtue of being written down*, lead that inference mechanism to behave in the way described.

Particular examples are easy to imagine. Thus consider a computational process designed to repair electronic circuits. One can imagine that it would be useful to have inference rules of the following sort: *"Unless you have been told that the power supply is broken. you should assume that it works"*, or, *"You should make checking capacitors your first priority, since they are more likely than are resistors to break down"*. Furthermore, it would be good to ensure that such rules could be modularly and flexibly added and removed from the system, without each time requiring surgery on the inner constitution of the inference engine. Though we are skirting close to the edge of an infinite regress, it is clear that something like this kind of protocol is a

natural part of normal human conversation. From an *intuitive* point of view it seems perfectly reasonable to say: *By the way, if you ever want to assume* p, *it would be sufficient to establish that you cannot prove its negation*. The question is whether we can make *formal* sense out of this intuition.

It is clear that the problem is not so much one of *what* to say, but of *how* to say it (to some kind of theorem-prover, for example) in a way that on the one hand does not lead to an infinite regress, and that on the other genuinely affects its behaviour. All sorts of technical question arise. It is not obvious what language to use, for example; or even to whom such a statement should be directed. Suppose, for example, that we were supplied with a monotonic natural-deduction based theorem prover for first order logic. Could we supply it with [1] as an ordinary material implication? Certainty not. At least in the form given above, it is not even a well-formed sentence. There are various ways we could *encode* it as a sentence—one way would be to use set theory, and to talk explicitly about the set of sentences derivable from other sentences, and then to say that if the sentence '¬p' is not in a certain set, then 'p' is. The problem is that while such a sentence might contribute to a *model* of the kind of inference procedure we desire, in any ordinary theorem prover simply adding it to the stock of implication that it has to work with would not *thereby cause the inference mechanism itself behave non-monotonically in the described way*. To do this would not be to construct a non-monotonic reasoning system, but rather to build a monotonic one prepared to reason about a non-monotonic one. While such a formulation might be of interest in the specification of the constraints a reasoning system must honour (a kind of "competence theory" for non-monotonic reasoning[4]), it would not help us, at least on the face of things, with the question of how a system using defaults might actually be deployed. Another option, of course, would be to build a non-monotonic

4. Reiter (1978), McDermott and Doyle (1978), Bobrow (1980).

inference engine from scratch, using expressions like [I] to constrain its behaviour, along the lines of abstract program specifications. But this would solve the problem by avoiding it—the whole question was how to use such comments on the reasoning procedure coherently *within the structures of the problem-specific application*.

Yet another possibility—one I will focus on for a moment—would be to design a more complex inference mechanism to react appropriately not only to sentences in the standard object language, but to meta-theoretic expressions of the form [I]. Although no system of just this sort has been demonstrated, such a program is readily imaginable, and various dialects of Prolog—perhaps most clearly the IC-PROLOG of Imperial College[5]—are best viewed in this light The problem with such solutions, however, is their excessive rigidity and inelegance, coupled with the fact that they do not really solve the problem in any case. What a Prolog user is given is not a unified or reflective system, but a pair of two largely *independent* formal systems: a basic declarative language in which facts about the world can be expressed, and a *separate* procedural language, through which the behaviour of the inference process may be controlled. Although the elements of the two languages are mixed in a Prolog program, they are best understood as separate aspects. One set (the structure of clauses, implications, and predicates, the identity of variables, and so forth) constitutes the *declarative* language, with the standard semantics of first-order logic. Another (the sequential ordering of the sentences and of the predicates in the premise, the "consumer" and "producer" annotations on the variables, the "cut" operator, and so forth) constitute the *procedural* language. Of course the flow of control is affected by the declarative aspects, but this is just like saying that the flow of control of an ALGOL program is affected by its data structures.

Thus the claim that to use Prolog is to "program in logic" is

5. Clark and McCabe (1979).

in my view misleading: rather, what happens is that one essentially writes programs in a new (and, as it happens, rather limited) control language, using an encoding of first-order logic as the declarative representation language. Of course this is a dual system with a striking fact about its procedural component: all conclusions that can be reached are guaranteed to be valid implications of prior structures in the representational field. As mentioned above, however, *dual-calculus* approaches of this sort seem ultimately rather baroque, and is certainly not conducive to the kind of reflective abilities we are after. It would be far more elegant to be able to say, *in the same language as the target world is described*, whatever it was salient to say about how the inference process was to proceed.

For example, to continue with the Prolog example, one would like to say both FATHER(BENJAMIN,CHARLES) and CUT(CLAUSE-13) or DATA-CONSUMER(VARIABLE-4) in one and the same language, with both subject to the same semantical and procedural treatment. The increase in elegance, expressive power, and clarity of semantics that would result are too obvious to belabour: just a moment's thought leads to one realise that only a single semantical analysis would be necessary (rather than two); the reflective capabilities could recurse without limit (Prolog and other dual-calculus systems intrinsically consist of just a single level); a meta-theoretic description of the system would have to describe only one formal language, not two; descriptions of the inference mechanism, would be immediately available, rather than having to be extracted from procedural code; and so forth.

This ability to pass coherently between two situations—in the reflective case to have the structures that normally control the interpretation process be fully and explicitly visible to (and manipulable by) the reasoning process, and in the other to allow the reasoning process to sink into them, so that they may take their natural effect as part of the tacit background

in which the reasoning process works—this ability is a particular form of reflection that I will call **procedural reflection** ("procedural" because I are not yet requiring that those structures at the same time *describe* the reasoning behaviours they engender; that is the larger task not yet taken on). Although ultimately limited, in the sense that a procedurally reflective calculus is by no means a fully reflective one, even this more modest notion is on its own a considerable subject of inquiry.

1b·iii Six General Properties of Reflection

Given the foregoing sketch of the task, it is appropriate to ask, before plunging into details, whether we can have any sense in advance of what form the solution might take. Six properties of reflective systems can be identified straight away—features that any ultimate solution should exhibit, however it ends up being structured and/or explained.

1b.iii.α Causal connection

First, the notion is one of self-reference, of a *causally-connected* kind, stronger than the notion explored by mathematicians and philosophers over much of the last century. What is needed is a theory of the causal powers required in order for a system's possession of self-descriptive and self-modelling abilities to *actually matter* to it—a requirement of substance, since full-blooded, actual behaviour is our ultimate subject matter, not simply the mathematical characterisation of formal relationships. A30

In dealing with computational processes, we are dealing with artefacts *behaviourally* defined, after all, unlike systems of logic, which are *functionally* defined abstractions that in no way behave or participate with us in the temporal dimension. Although any abstract machine of Turing power can provably model any other—including itself—there can be no sense in which such self-modelling is even *noticed* by the underlying

machine (even if we could posit an *animus ex machina* to do the noticing). If, on the other hand, our aim is to build a computational system of substantial reflective power, we will have to build something that is affected by its ability to "think about itself." This holds no matter how accurate the self-descriptive model may be; you simply cannot afford simply to reason about yourself as disinterestedly and inconsequentially as if you were someone else.

Similar requirements of *causal connection* hold of human reflection. Suppose, for example, that after taking a spill into a river I analyse my canoeing skills and develop an account of how I would do better to lean downstream when exiting an eddy. Coming to this realisation is useful just in so far as it enables me to improve. If I merely smile in vacant pleasure at an *image* of an improved me, but then repeat my ignominious performance—*if in other words my reflective contemplations have no effect on my subsequent behaviour*—then my reflection will have been in vain. It is crucial, in other words, to make the move from description to reality. In addition, just as the *result* of reflecting has to affect *future* non-reflective behaviour, so does *prior* non-reflective behaviour have to be accessible to reflective contemplation; one must equally be capable of moving from reality to description. It would have been equally futile if, when I initially paused to reflect on the cause of my dunking, I had been unable to remember what I had been doing just before I capsised.

In sum, the relationship between reflective and non-reflective behaviour must be of a form such that both information and effect can pass back and forth between them. These requirements will impinge on the technical details of reflective calculi: we will have to strive to provide sufficient connection between reflective and non-reflective behaviour so that the right causal powers can be transferred across the boundary, without falling into the opposite difficulty of making them

so interconnected that confusion results. (An example is the issue of providing continuation structures to encode control flow: we will provide *separate* continuation structures for each reflective level, to avoid unwanted interactions, but we will also have to provide a way in which a designator of the lower level continuation can be bound within the environment of the higher one, so that a reflective program can straightforwardly refer to the continuation of the process below it).

The interactions between levels can grow rather complex. Suppose, to take another example, that you decide at some point in your life that whenever some type of situation arises (say, when you start behaving inappropriately in some fashion), that you will pause to calm yourself down, and to review what has happened in the past when you have let your basic tendencies proceed unchecked. The dispassionate fellow that you must now become is one that embodies, in their current and on-going being, a decision made now *at some future point to reflect*. Somehow, without acting in a self-conscious way from now until such a circumstance arises, you have to make it true that when the situation *does* arise, you will have left yourself in a state that will cause the appropriate reflection to happen *then*. By the same token, in the technical formalisms we design, we have to provide the ability to descend ("drop down") from a reflected state to a non-reflected one, having left the base level system in such a state so that, when certain situations occur in the future, the system will automatically reflect *at that point*, and thereby obtain access to the reasons that were marshalled in support of the original decision. A31

*1b.iii.*β *Theory relativity*

Second, reflection has something, although just what remains to be seen, to do with self-*knowledge*, as well as with self-*reference*—and knowledge, as has often been remarked, is inherently theory-relative (in a way that pure self-reference is not).

Just as one cannot interpret the world except through using the concepts and categories of a theory, one cannot reflect on one's self except in terms of the concepts and categories of a theory of self. Furthermore, as is the case in any theoretical endeavour, the phenomena under consideration under-determine the theory that accounts for them, even when all the data are to be accounted for. In the more common case, when only parts of the phenomenal field are to be treated by the theory, an even wider set of alternative theories emerge as possibilities. In other words, *when you reflect on your own behaviour, you must inevitably do so in a somewhat arbitrary theory-relative way*.

One of the mandates must be set for any reflective calculus, therefore, is that it be provided, represented in its own internal language, with an (in some appropriate sense) complete theory of how it is formed and of how it works. Theoretical entities may be posited by this account that facilitate an explanation of behaviour, even though those entities cannot be claimed to have a theory-independent ontological existence in the behaviour being explained. 3Lisp will be provided with a "theory" of 3Lisp in 3Lisp, for example, reminiscent of the metacircular interpreter demonstrated in McCarthy's original report[6] and in the reports of Sussman and Steele[7]—but causally connected in novel ways. In providing this primitively supported reflective model, I adopt a standard account, in which a number of notions commonly used to describe Lisp play a central role—such as that of an *environment*, just mentioned, and a parallel notion of a *continuation*. In spite of their familiarity, however, these have historically remained Lisp-*external notions, being used only to describe (and model) Lisp, rather than figuring as* first-class objects *internal to the language* in any direct sense. It is impossible in a *non-reflective* Lisp to define a predicate true only of environments, since environments as such do not exist in such dialects. Because its reflective ca-

6. McCarthy et al. (1965).
7. Sussman and Steele (1975); Steele and Sussman (1978a).

pacities are defined in terms of an environment and continuation-based theory, the notion of an environment becomes language-*internal to 3Lisp*—with environment representing structures being passed around as first-class entities.

There are other possible Lisp theories, some of which differ substantially from the one I have chosen. For example, it is possible to replace the notion of environment altogether (note that the λ-calculus is explained without any such device). If a reflective dialect were defined in terms of this alternative theoretical account (call such a language 3Lisp′), environments would no longer be a language internal concept. It would be likely, however, that this theory would posit other kinds of object, or other notions (such as α- and β-reduction), and in virtue of being reflective in 3Lisp′ those notions would become language-internal. In order to reflect you have to use *some* theory and its associated theoretical concepts and entities.

1b.iii.γ 'Reflective' vs. 'reflexive'

The third general point about reflection regards its name. I have deliberately chosen the term 'reflective,' as opposed to 'reflexive,' since there are various senses (other recent research reports not withstanding[8]) in which no computational process, in any sense I can understand, can succeed in narcissistically thinking *about the fact that it is at that very instant thinking about itself thinking about itself thinking*...and so on and so on, like a transparent eye in a room full of mirrors. The kind of A32 reflecting I will consider—the kind that 3Lisp demonstrates how, technically, to define, implement, and control—requires that in the act of reflecting the process "take a step back" in order to allow the interpreted process to consider what it was just up to from a different vantage point, to bring into view symbols and structures that describe its state "just a moment earlier." From the mere fact of a system's having a name for itself it does not follow that the system thereby automatically

8. Greiner and Lenat (1980), Genesereth and Lenat (1980).

acquires the ability to *focus on its current instantaneous self*, for in the process of "stepping back" or reflecting, the "mind's eye" moves out of its own view, being replaced by an (albeit possibly complete) account of itself. (Though this description is surely more suggestive than incisive, the technical work to be presented will help to make it precise.)

1b.iii.δ Fine-grained control

Fourth, in virtue of reflecting a process can always obtain a finer-grained control over its behaviour than would otherwise be possible. What was previously an inexorably atomic stepping from one state to the next is opened up so that each move can be analysed, countered, and so forth—and also be broken down into constituent parts. As we will see in detail, in this way reflective powers give a system a far more subtle and more catholic—if less efficient—way of reacting to a world. The requirement here is the usual one: for what was previously implicit to be made explicit, albeit in a controlled and useful way, without violating the ultimate truth that not everything can be made explicit in a finite mechanism. This ability enables a system designer to satisfy what might otherwise be taken to be incompatible demands: (i) the provision of a small and elegant kernel calculus, with crisp definition and strict behaviour; and at the same time (ii) the ability for the user (by using reflection) to be able to modify or adjust the behaviour of this kernel in peculiar or extenuating circumstances. One of reflection's great powers is that it allows such simplicity and flexibility to be achieved simultaneously.

1b.iii.ε Partial detachment

This leads to the fifth general comment, which is that the ability to reflect never provides a complete separation, or an utterly objective vantage point from which to view either oneself or the world. No matter now reflective any given system or

person may be, it remains a truism that there is ultimately no escape from being the person in question. Though as the dissertation proceeds I will increasingly downplay any connection between the formal work presented here and human abilities, it is still perhaps helpful to say that the kind of reflection to be presented here is closer to what is known as *detachment* or *awareness* than it is to a strict kind of self-objectivity (this is why I have been and will remain systematically imprecise about whether *reflection* is fundamentally a way to think about oneself or a way to think about the world).

The environment example just mentioned provides an illustration in a computational setting. As we will see in detail, the environment in which are bound the symbols that a program is using is, at any level, merely part of the embedding background in which the program is running. The program operates within that background, dependent on it but—in the normal (unreflective) course of events—unable to access it explicitly. The operation of reflecting makes explicit what was just implicit: it renders visible what was tacit, what was in the background. In doing so, however, a new background fills in to support the reflective deliberations. Again, the same is true of human reflection: you and I can interrupt our conversation in order to sort out the definition of a contentious term, but—as has often been remarked—we do so using other terms. Since language is our inherent medium of communication, we cannot step out of it to view it from a completely independent vantage point. Similarly, while the systems I will show how to build can at any point back up and *mention* what was previously *used*, in doing so more structured background will come into implicit use.

This lesson, of course, has been a major one in philosophy at least since Peirce; certainly Quine's famous comment about Ncurath's boat holds as true for the systems we design as it does for us designers.[9] A33

9. Quine (1953a), p. 79 in the 1963 edition.

1b.iii.ζ Kernel requirements

Sixth and finally, the ability to reflect is something that must be built into the heart or kernel of a calculus. There are theoretically demonstrable reasons why reflective powers cannot be "progrrammed up" as an addition to a calculus (though one can of course *implement* a reflective machine in a non-reflective one: the difference between these two must always be kept in mind). The reason for this claim is that, as discussed in the first comment, *being reflective* is a stronger requirement on a calculus than simply *being able to model the calculus in the calculus*, something of which any machine of Turing power is capable (this is the "making it matter" that was alluded to above). This will be demonstrated in detail; the crucial difference, as suggested above, comes in connecting the self-model to the basic interpretation functions in a causal way, so that (for example and very roughly) when a process "decides to assume something," it can thereby in fact assume it, rather than simply constructing a model or self-description or hypothesis that *represents itself* as assuming it. As well as "backing up" in order to reflect on its thoughts or operations, in other words, a reflective process must be able to "drop back down again" to consider the world directly, in accord with the consequences of those reflections. Both parts of this involve a causal connection between the explicit programs and the basic workings of the abstract machine, and such connections cannot be "programmed into" a calculus that does not support them primitively.

1b·iv Reflection and Self-Reference

At the beginning of this section I said that my investigation of reflection in general would primarily concern itself, because of operating under the knowledge representation hypothesis, with the *self-referential* aspects of reflective behaviour. There has been in the last century no lack of investigation into self-

referential expressions in formal systems, especially since it has been exactly in these areas where the major results on paradox, incompleteness, undecidability, and so forth, have arisen. It is therefore helpful to compare the present enterprise with these theoretical precursors.

Two facets of the computational situation show how very different our concerns here will be from these more traditional studies. First, although I do not formalise this, there is no doubt in my work that I consider the locus of *referring* to be *an entire process*, not a particular expression or structure (especially not a *solitary* expression or structure). Even though I will posit declarative semantics for individual expressions, I will also make evident the fact that the designation of any given expression is a function not only of that expression itself, but also of the state of the processor *at the point of that expression's use*. And to the extent that "use" is even a coherent term for symbolic activity, it is the processor that uses the symbol; the symbol does not use itself. To the extent that we want a system to be self-referential, then, we want *the process as a whole* to be able to refer, to first approximation, *to its whole self*, although in fact this usually reduces to a question of it referring to some of its own ingredient structure.

Achieving this goal is not only not met by providing the system with self-referential structure, but even more strongly, I avoid such self-referential structures entirely, exactly to avoid many of the intractable (if not inscrutable) problems that arise in such cases. Because of its λ-calculus base, it is perfectly possible in 3Lisp to construct apparently self-designating expressions (at least up to type-equivalence: token self-reference is more difficult). But from a practical point of view the system of levels I will embrace will by and large exclude such local self-reference from our consideration. Truly self-referential expressions, such as *This sentence is six words long*, are unarguably odd, and certain instances of them, such as the clichéd

This sentence is false, are undeniably problematic (strictly speaking, of course, the sentence "This sentence is six words long" *contains* a self-reference, but is not itself self-referential; however we could use instead the composite term "this five word noun phrase"—though it is not as immediately evident that this leads to trouble). None of these truths impinge particularly on our quite different concerns.

The second comment (illustrating how different 3Lisp and procedural calculi are from mathematical and logical studies of self-reference) is this: in traditional formal systems, the *actual reference* relationship between any given expression and its referent (whether that referent is itself or a distal object) is mediated by an externally attributed semantical interpretation function. The sentence "*This sentence is six words long*" does not actually *refer*, in any causal full-blooded sense, to anything; rather, we English speakers *take it* to refer to itself. The reference relation connecting that sentence in its role as sign, and that same sentence in its role as referent or significant, flows through us.

As I said in the previous section in the discussion of causal connection, in constructing reflective computational systems it is crucial that the causal mediation *not be* deferred through an external observer. Reflection in a computational system *has to be causally connected internally*, even if the *semantical* understanding of that causal connection is externally attributed. For example, in 3Lisp there is a primitive relationship that holds between a certain kind of symbol, called a *handle* (a canonical form of meta-descriptive rigidly-designating name) and another symbol that, semantically, each handle designates. I.e., handles are the 3Lisp structural form of *quotation*. Suppose that H_1 is a handle, and that S_1 is some structure that H_1 refers to. Strictly speaking, there is an internal structural relationship between H_1 and S_1, which we, as external semantical attributors, take in addition to be a *reference* relationship.

Until we can construct computational systems that are what I have called semantically original. the *semantical* import of that relationship will always remain externally mediated. But the *causal* relationship between H_1 and S_1 *must be* internal: otherwise there would be no way for the internal computational processes to treat that relationship in any way that mattered. A34

This may be clearer if put a bit more formally. Suppose that φ is the externally attributed semantical interpretation function, and that ζ is the primitive, effective structural function that relates handles to those structures we call their referents. It is ζ that will allow the processor to produce or obtain causal access to a structure S given that H is its handle. Thus in the prior example, it is true both that $\varphi(H_1)=S_1$, due to our external semantical attribution of reference to H, and that $\zeta(H_1)=S_1$. More generally, we know, given the 3Lisp architecture, that:

$$\forall H,S\ [\![\text{HANDLE}(H) \wedge \zeta(H)=S]\!] \equiv [\![\varphi(H)=S]\!] \qquad [2]$$

However, though in some sense it is strictly true, this equation in no way reveals the *structure* of the relationship between φ and ζ; it merely states their extensional equivalence. More revealing of the fact that I take the relationship between handles and referents to be a reference relation (if I may wantonly reify relationships for a moment) is the following:

$$\varphi(\zeta)=\varphi \qquad [3]$$

Of, rather, since not all symbols are handles. as:

$$\varphi(\zeta) \subset \varphi \qquad [4]$$

The requirement that reflection *matter,* to summarise, is a crucial facet of computational reflection—one without precedent in pre-computational formal systems. What is striking is that the mattering cannot be derived from the semantics, since it would appear that mattering—which requires a real causal connection—is a *precursor* to semantical originality, not some-

thing that can *follow* semantical relationships. Put another way, in the inchoately semantical computational systems I am trying to build, the reference relationships between internal meta-level symbols and their internal referents (the semantical relationships crucial in reflective considerations) may have to be causal in *two distinct ways:* once mediated by us, who attribute semantics to those symbols in the first place, and a second time internally, so that the appropriate causal behaviour, to which we attribute semantics, can be engendered. On that day when we succeed in constructing semantically original mechanisms, those two presently independent causal connections may merge; until then we will have to content ourselves with *causally original* but *semantically derivative* systems. The reflective dialects I will propose will all be of this form.

1c A Process Reduction Model of Computation

I next want to sketch the model of computation on which the analysis and design of 3Lisp will depend.

I take **processes** to be the fundamental subject matter; though I will not define the concept precisely, we can assume that a process consists approximately of a connected or coherent set of events through time. The reification of processes as objects in their own right—composite and causally engendered—is a distinctive, although not distinguishing, mark of computer science. Processes are inherently temporal, but not otherwise physical: they do not have spatial extent, although A35
they *must* have temporal extent Whether there are more abstract dimensions in which it is appropriate to locate a process is a question I will sidestep; since this entire characterisation is by way of background for another discussion, I will rely more on examples and on the uses to which we put these objects than on explicit formulation.

I will depict processes as in figure 1, on the next page. The boundary of the icon is intended to signify the *boundary* or

surface of the process itself, taken to be the interface between the process and the world in which it exists (I take objectifying processes to involve "carving them" out of a world in which they can then be said to be embedded). Thus the set of events that collectively form the behaviour of a coherent process in a given world would consist of all events on the surface of this abstract object. This set of events could be more or less specifically described: we might simply say that the process had certain gross input/output behaviour (with "input" and "output" being defined as a certain class of surface perturbation—an interesting and non-trivial problem), or we might account in tine detail for every nuance of the process' behaviour, including the exact temporal relationships between one event and the next, and so forth.

Process

Figure 1

It is crucial to distinguish these more and less fine-grained accounts of the *surface* of a process, on the one hand—its behavioural interface or interactions with its environment—from compositional accounts of its interior, on the other. That a process has such an "interior" is again a striking assumption throughout computer science: the role of what in computer science are universally called **interpreters,** though I myself A36 will use the term **processors**, is a striking example. Suppose for instance that one were interact with a so-called "Lisp-based editor." It is standard to assume that the Lisp interpreter (processor) is an *ingredient process* within the process with which you interact: moreover, it is understood to be the locus of *anima* or *agency* inside your editor process, that in turn supplies the temporal action or activity in the editor itself. That is, of all the interior ingredients constituting the editor, only the interpreter (processor) is understood to be *active*; all other components—specifically, the "editor program" and any associated data structures—will be static or at least passive, at

least at this level of abstraction. Yet the one active ingredient (interior) process *never appears as the surface of the editor*: no user interaction with the editor (via the keyboard, say) is itself directly an interaction with the Lisp processor. Rather, the Lisp processor, in conjunction with some appropriate (passive) Lisp *program*, together engender the behavioural surface with which the user interacts.

Computer science has studied a variety of such architectures—or classes of architecture; here I will briefly mention just two, but will then focus, throughout the rest of the dissertation, on just one. Every computational process, I will assume (I will take on the question of which processes we are disposed to call computational in a moment), has within it at least one other process, which, singly or collectively, supplies the animate agency of the overall constituted process.

I will call this model a **process reduction** model of computation. since at each stage of *computational reduction* a given process is reduced in terms of constituent symbols and other processes. There may be more than one internal process (in what are known as *parallel* or *concurrent* processes), or there may be just a single one (known as *serial* processes). Reductions of processes that do not posit an interior process as the source of the agency I will consider to be outside the realm of computer science proper—though of course some such reduction must at some point be accounted for, if the engendered process is ever to be realised. I will view these alternatives forms of reduction—from process to, say, behaviours of physical mechanism—to fall more within physics or electronics (or perhaps computer engineering) than within computer science *per se*. What is critical is that at some stage in a series of computational reductions this leap from the domain or processes to the domain of mechanisms be taken, as for example in the explaining how the behaviour of a set of logic circuits constitutes a processor (interpreter) for the microcode

of a given computer. Given this one account of what may reasonably be called the **realisation** of a computational process, an entire hierarchy of processes above it may obtain indirect realisation through a series of process reductions of the above form. For example, if that microcode processor interprets a set of instructions that are the program for a macro machine (say, a CPU), then a macro processor—an interpreter (processor) for the resulting "machine language" may be said to exist. Similarly, that macro machine may in turn interpret (process) a machine language program that implements SNOBOL: thus by two stages of "process composition" (i.e., the inverse of process reduction) a SNOBOL processor is also realised.

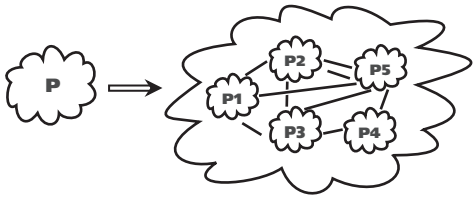

Figure 2

In order to make this talk of processors and so forth a little clearer, it helps to diagram two different forms of process reduction: what I will call *[parallel] reduction* and *[serial] reduction*. Taking '⇒' to mean "reduces to," figure 2 depicts [parallel] reduction, by showing that process P reduces to a set of five interior processes ($P_1 \ldots P_5$). How these processes [interact] I will not here say: I merely assume that those five ingredient processes do interact in some fashion, so that taken as a composite unity their total behaviour is (i.e., can be "interpreted" as) the behaviour of the thereby constituted process. Responsibility for the surface of the total process P is assumed to be shared in some way amongst the five ingredients. Examples of this sort of reduction may be found at any level of the computational spectrum—from metaphors of disk-controllers communicating with bus mediators communicating with central processors, to the message-passing metaphors in such Artificial Intelligence languages as ACTI and Smalltalk and so forth.[10]

A37 A38 A39

10. For references on the message-passing metaphor, see Hewitt et al. (1974) *(cont'd)*

[Parallel] reductions will receive only passing mention in this dissertation; I discuss them only in order to admit that the model of reflection that I will propose is not (at least at present) sufficiently general to encompass them. Instead I will focus instead on the more common model that I am calling **[serial] reduction**, pictured in figure 3. In such cases the overall process is composed of what I will call a **processor** and a **structural field**. The former ingredient is the locus of active agency; as already mentioned, it is what is typically called an 'interpreter,' but from here on I will avoid that term (or when using it, do so within quotation marks), because of its confusion with semantical notions of interpretation from the declarative tradition (I will have much more to say about this confusion in [dissertation] chapter 3). The latter ingredient is intended to include both the program or the program's data structures (or both); it is often taken to consist of a set of *symbols*, although that term is so semantically loaded that for the time being I will avoid it as well. A40

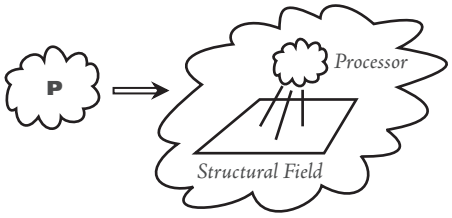

Figure 3

One benefit of the [serial] model of process reduction is that it can be used to understand both language design and the construction of particular programs. For example, we can characterise Fortran in its terms, by positing a Fortran "processor" that computes over (examines, manipulates, constructs, reacts to, and so forth) elements of the Fortran structural field, which includes primarily an ordered sequence of Fortran instructions, FORMAT statements, arrays, etc. Suppose you were to set out to develop a Fortran "program" (really: process) to manage your financial affairs—which for discus- A41

...

and Hewitt (1977); for ACT1 see Lieberman (1987); for Smalltalk see Goldberg (1981), Ingalls (1978).

sion I will call *Chequers*. To do this, you would specify a set of Fortran data structures, and design a process to interact with them. In terms of the model, those data structures—the tables that list current balances, recent deposits, interest rates, currency conversion factors, and so on—would constitute the structural field of the first [serial] process reduction of Chequers. The "program" (i.e., process) you design to interact with these data base I will simply call P_c. Thus the first Chequers [serial] reduction would be pictured in the model as depicted in figure 4.

We are assuming, however, that P_c is specified by a Fortran program. P_c is not itself that program—or any program, for that matter; P_c is a *process*, and programs are static, requiring interpretation by a processor in order to engender processes or behaviour. Rather, P_c can itself be understood in terms of a second [serial] reduction, of the program C that, when processed by the Fortran processor, yields process P_c as a result. In toto, that is, the development of Chequers involves have a double [serial] reduction, depicted in figure 5.

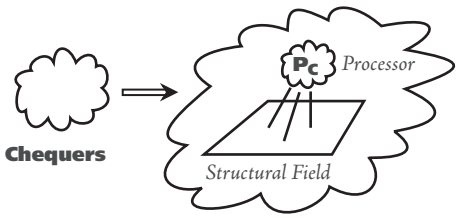

Figure 4

A host of questions would have to be answered before this model could be made precise (before, for example, one could develop anything like an adequate mathematical treatment of these intuitions). For example, the data structures in the foregoing example themselves have to be implemented in Fortran as well. However to fill out the model just a little, we can suggest how we might, in these terms, define a variety of commonplace terms of art of computer science.

First, I take the computer science term 'interpreter' (which,

to repeat, I will call a *processor*) to be used in the following way:

> **Interpreter:** *A process that is the interior process in an [serial] reduction of another interior process.*

For example, the process P_c developed in the course or implementing Chequers is not an interpreter, on this definition, because although it is an ingredient process (it is not, in particular, Chequers itself, but rather interior to Chequers), it is nevertheless interior only singly. The process thereby constituted—viz., Chequers—is not itself an interior process. On the other hand, it is legitimate to call the process that "interprets" (i.e., processes) Lisp programs an interpreter, because Lisp programs are structural field arrangements that engender other interior processes that work over data structures so as to yield yet other processes.

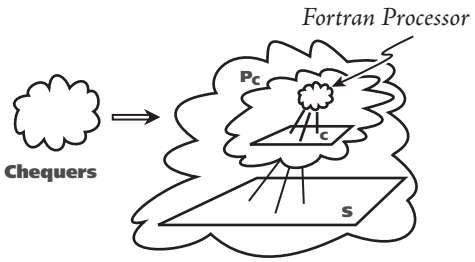

Figure 5

Second, I would argue that we use "compilation" as follows:

> **Compilation:** *The transformation or translation of a structural field arrangement* S_1 *to another structural field arrangement* S_2*, in such a way that the surface behaviour of the process* Q_1 *that would result from the processing of* S_1 *by some processor* P_1 *is equivalent—modulo some appropriate equivalence metric—to the surface behaviour of the process* Q_2 *that would result from the processing of* S_2 *by some other processor* P_2.

For example, I spoke above about a Fortran "processor," but of course such a processor is rarely if ever realised. Rather, Fortran programs are typically compiled—usually into some form of machine language. Consider the compiler that com-

piles Fortran into the machine language of the IBM 360. Then the compilation of a particular Fortran program c_F into an IBM 360 machine language program c_{360} would be *correct* just in case the surface of the process that would result from the processing of c_F by the (hypothetical) Fortran processor would be equivalent to the process that will actually result by the processing of c_{360} by the basic IBM 360 machine language processor.

In sum, compilation is defined relative to two [serial] reductions, and is mandated only to ensure equivalence, modulo an appropriate metric, of resulting process surfaces.

Third, by 'implementation' I take it that we refer to two kinds of construction.

> **Process Implementation (i.e., programming):** *The construction of a structural field arrangement* s *for some processor* p *such that the surface of the process that results from the processing of* s *by* p *yields the desired behaviour—i.e., desired process* Q.

More interesting is to implement a **computational language**. In terms of the model, we can characterise (serial) computer languages as follows:

> **Computational Language:** *The architecture of a structural field and a behaviourally specified processor for it, in which are specified both possible arrangements or configurations of the field, and the behaviour that would result from the processing of them by the specified processor.*

In terms of this definition, we can characterise the *implementation* of a language:

> **Language Implementation:** *The provision of a process* p *that can be [serially] reduced to the structural field and interior processor of the language being implemented.*

To implement Lisp, in other words, all that is required is the provision of a process that *behaviourally* appears to be a constituted process consisting of the Lisp structural field and the interior Lisp processor. Thus I am completely free of any actual commitment as to the reality, if any, of the implemented field. A42

Typically, one language is implemented in another by constructing some arrangement or set of protocols on the data structures of the implementing language to encode the structural field of the implemented language. and by constructing a program in the implementing language that, when processed by the implementing language's processor, will yield a process whose surface can be taken as a processor for the interpreted language, with respect to that encoding of the implemented language's structural field. (By a *program* I refer to a structural field arrangement *within an interior processor*—i.e., to the inner structural field of a double reduction—since programs are structures that are interpreted to yield processes that in turn interact with another structural field (the data structures) so as to engender a whole constituted behaviour.)

Finally, it is straightforward to imagine how this model could be used in cognitive theorising. A **weak** computational model of some mental phenomenon or behaviour ψ would be a computational process that was claimed to be superficially equivalent to ψ (as always: modulo some equivalence metric). Note that surface equivalence of this sort can be arbitrarily fine-grained. Just because a given computational model predicts the most minute temporal nuances revealed by click-stop experiments and so forth, that does not imply that anything other than surface equivalence has been achieved In contrast, a **strong** computational model would posit not only surface but *interior architectural structure*. Thus for example Fodor's recent claim of mental modularity[11] is a coarse-grained but strong claim: he suggests that the dominant or overarching computational reduction of the mental is closer to a [parallel] than to a [serial] reduction.

11. Fodor (forthcoming).

* * *

This has been the briefest of sketches of a substantial subject. Ultimately, it should be formalised into a generally applicable and mathematically rigorous account. In this dissertation I will merely use its basic conceptual structure to organise the analysis, and will also base the 3Lisp architecture on it. Even for these purposes, however, it is important to identify three properties that all structural fields must manifest. A43

First, over every structural field there must be defined a **locality** metric or measure—since (in concert with physical constraint) *the interaction of a processor with a structural field is always constrained to be locally continuous*.

Informally, one can think of the processor looking at the structural field with a pencil-beam flashlight—able to see and react only to what is currently illuminated (more formally, the behaviour of the processor must always be a function only of its internal state plus the current single structural field element under investigation). Why it is that the well-known joke about a COME-FROM statement in Fortran is funny, for example, can be explained only because this it violates this local accessibility constraint (it is otherwise perfectly well-defined). Note as well that in logic, the λ-calculus, and so forth, no such locality considerations come into play. In addition, the measure space yielded by this locality metric need not be symmetrical, as Lisp demonstrates; from the fact that A is accessible from B it does not follow that B must be accessible from A. A44

Second—and this is a major point, with which we will need to grapple considerably in our considerations of semantics—structural field elements *are taken to be significant* or *meaningful*. This is why we tend to call them *symbols*. In particular, i will count as *computational* only those processes consisting of ingredient structures and events to which we, as external observers, attribute semantical value or import.

The reason I do not consider a car to be a computer, even if I am tempted to think of its electronic fuel injection module computationally, hinges explicitly on this issue of semantical attribution. The main components of a car we understand in terms of mechanics—forces, torques, plasticity, geometry, heat, combustion, and so on. These are not "interpreted" or semantical notions; or to put the same point another way, explaining a car does not require positing any externally attributed semantical interpretation function in order to make sense of a car's inner workings. With respect to a computer, however—whether abacus, calculator, electronic fuel injection system, or a full-scale digital computer—the best explanation is exactly in terms of the interpretation of the ingredients, even though the machine itself is not allowed access to that interpretation (for fear of violating the strictures of mechanism). Thus while I may know that the arithmetic logical unit in my machine works in such and such a way, I nevertheless "understand" its workings in terms of addition, logical operations, and so forth, all of which speak about the interpretations of its parts and workings, rather than speaking about them directly. In other words the proper use of the term "computational" is as a predicate on explanations, not on artefacts. A45

The third constraint follows directly on the second: in spite of this semantical attribution, the interior processes of a computational process must interact with these structures and symbols and other processes *in complete ignorance and disregard of any this externally-attributed semantical weight.* This is the substance of the claim that computation is **formal** symbol manipulation—that computation has to do with the interaction with symbols solely in virtue of their spelling or shape. We computer scientists are so used to this formality condition—this requirement that computation proceed "syntactically"—that we are liable to forget that it is a major claim, and are in danger of thinking that the simpler phrase "symbol ma- A44

nipulation" *means* formal symbol manipulation. Nevertheless, part of the semantical reconstruction to be undertaken here will rest on a claim that, in spite of its familiarity, we have not taken semantical attribution seriously enough.

A book should be written on all these issues; I mention them here only because they will play an important role in the upcoming reconstruction of Lisp. There are obvious parallels and connections to be explored, for example, between this external attribution of significance to the ingredients of a computational process, and the issue of what would be required far a computational system to be semantically original in the sense discussed at the beginning of the previous section. This is not the place for such investigations; but as §1·d and [dissertation] chapter 3 will make clear, below, this attribution of significance to Lisp structures must be part of the full declarative semantics for Lisp. The present moral is merely that, although including such interpretation within the scope of an account of a language's semantics has not (to my knowledge) been done before, the attribution of semantic interpretation itself is neither something new, nor something specific to Lisp's circumstances. Externally attributed (declarative) significance is a foundational part of computing, even if not yet fully recognised in computer science.

1d The Rationalisation of Computational Semantics

From even the few introductory sections that have been presented so far, it is clear that semantical vocabulary will permeate the upcoming analysis. In discussing the Knowledge Representation and Reflection hypotheses, I talked of symbols that *represented* knowledge about the world, and of structures that *designated* other structures. In the model of computation just presented, I said that the attribution of *semantic signifi-*

cance to the ingredients of a process was a distinguishing mark of computing. Informally, no one could possibly understand Lisp without knowing that the atom T stands for truth, and NIL for falsity. If we subscribe to the view that computer science is about *formal* symbol manipulation, we admit not only that the subject matter involves *symbols*, but also that any computations over them must occur in ignorance of their semantical weight (you cannot treat a non-semantical object, such as an eggplant or a waterfall, *formally*, unless you first, non-standardly, set it up as a symbol; the mere use of the predicate 'formal' assumes that its object is significant, or has been attributed significance, even if on the side). Even at the very highest levels, when we say that a process—human or computational—is reasoning *about* a given subject, or reasoning *about* its own thought processes, we implicate semantics, since the term 'semantics' can (at least in part) be viewed as merely a fancy word for *aboutness*.

It is therefore necessary for me to add to last section's account of processes and process reduction a corresponding accounting of the semantical assumptions I will make and techniques I will use, and to make clear what I mean when we say that I will subject computational dialects to semantical scrutiny.

1d·i Pre-Theoretic Assumptions

When we engage in semantical analysis, I do not take it to be our goal simply to provide a mathematically adequate specification of the behaviour of one or more procedural calculi that would enable us, for example, to prove that programs will meet some specification of what they were designed to do. That is: by "semantics" I do not simply mean a mathematical formulation of the properties of a system, formulated from a meta-theoretic vantage point. (Unfortunately, in my view, in some writers the term seems to be acquiring this weak connotation.) Rather, A47

I take semantics to have fundamentally to do with meaning and reference and so forth—whatever they come to—as paradigmatically manifested in human thought and language (as was mentioned in §1b·i). I am therefore interested in semantics for two reasons: first, because, as I said at the end of the last section, all computational systems are marked by external semantical attribution; and second, because semantics is the study that will reveal what a computational system is reasoning *about*, and a theory of what a computational process is reasoning about is a pre-requisite to a proper characterisation of reflection.

Given this agenda, I will approach the semantical study of computational systems with a rather precise set of guidelines. In particular, I will require that any subsequent semantical analyses answer to the following two requirements, emerging from the two facts about processes and structural fields laid out at the end of section:

1. They should manifest the fact that we understand computational structures in virtue of attributing to them semantical import;
2. They should make evident that, in spite of such attribution, computational processes are *formal*, in that they must be defined over structures independent of their semantical weight.

These two principles alone entail the requirement of a double semantics, since the attributed semantics mentioned in the A48
first premise includes not only a pre-theoretic understanding of *what happens* to computational symbols, but also a *pre-computational* intuition as to what those symbols *stand for*. It follows that we will have to make clear the *declarative* semantics of the elements of (in our case) the Lisp structural field, as well as establishing their *procedural* import

I will explore these results in more detail below, but in bare

outlines the argument is straightforward. Most of the results are consequences of the following basic tenet (relativised here to Lisp, for perspicuity, but the same would hold for any other calculus):

> *What Lisp structures mean is not a function of how they are treated by the Lisp processor. Rather, how they are treated is a function of what they mean.*

For example, I take it that the Lisp expression "(+ 2 3)" evaluates to "5" for the undeniable reason that "(+ 2 3)" is understood as a complex *name* of the number that is the successor of four. We arrange things—we define Lisp in the way that we do—so that the numeral 5 is the value *because we know in advance what* (+ 2 3) *stands for.* To borrow a phrase from Barwise and Perry, this reconstruction is an attempt to "*regain our semantic innocence*"—an innocence that still permeates traditional formal A49 systems (logic, the λ-calculus, and so forth), but that has been lost in the attempt to characterise the so-called "semantics" of computer programming languages.

That "(+ 2 3)" designates the number five is self-evident, as are many other examples on which I will begin to erect my denotational account. I have also already alluded to the equally unarguable fact that (at least in certain contexts) T and NIL designate Truth and Falsity. Similarly, it is commonplace use the term "CAR" as a *descriptive function* to designate the first element of a pair, as for example in the English sentence "I noticed that the CAR of that list is the atom LAMBDA." The important point is that, in that English sentence, the phrase "CAR of that list" occurs as a *name* or a *designator*—not as a *procedure call*. Nothing *happens*, when I say it; it is not *executed*. It is merely a way of *pointing* to something—to the first element of the list pointed to by the ingredient phrase 'that list.' Similarly, it is hard to imagine an argument against the idea that "(QUOTE X)" designates X—in contrast to the claim, which is also often A50

heard, that does not speak at all about naming or designation, but only about procedural treatment: that QUOTE is a function that *holds off the evaluator*.

In sum, the moral is not so much that formulating the declarative semantics of a computational formalism is difficult, as that it must be recognised as an important thing to do.

1d·ii Semantics in a Computational Setting

In the most general form that I will use the term *semantics*,[12] a semantical investigation aims to characterise the relationship between a **syntactic** domain and a **semantic** domain—a relationship typically studied as a mathematical function mapping elements of the first domain into elements of the second. I will call such a function an **interpretation** function (it was in order to be able to talk about this function, which must be sharply distinguished from what is called an 'interpreter' in computer science, that I switched to the term *processor*). Schematically, that it, as shown in figure 6, the function φ is taken to be an interpretation function from S to D.

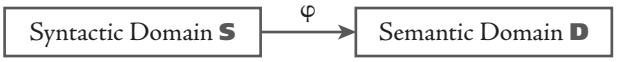

Figure 6

In a computational setting, this simple situation is made more complex because we are studying a variety of interacting interpretation functions. In particular, figure 7 identifies the relationships between the three main semantical functions that will permeate the analysis of 3Lisp. θ is the interpretation function mapping notations into elements of the structural field, φ is the interpretation function making explicit our attributed semantics to structural field elements, and ψ is the function formally computed by the language processor. ω will be explained below; it is intended to indicate a φ-semantic characterisation of the relationship between s_1 and s_2, whereas

12. See the postscript, however, where I in part disavow this fractured notion of syntactic and semantic domains. A51

ψ indicates the formally computed relationship—a distinction similar, as I will soon argue, to that between the logical relationships of *derivability* (⊢) and *entailment* (⊨). A52

The names have been chosen for mnemonic convenience: 'ψ' by analogy with *psychology,* since it is a study of the internal relationships between and among symbols, all of which are within the machine ('ψ' in this sense is meant to signify psychology *narrowly construed,* in the sense of Fodor, Putnam, and others[13]). The label 'φ', on the other hand, chosen to suggest *philosophy,* signifies the relationship between a set of symbols and the world. By analogy, suppose we were to accept the hypothesis that people represent or encode English sentences in an internal mental language called mentalese (suppose, in other words, that we accept the hypothesis that our minds are computational processes). If you say to me "A composer who died in 1750" and I respond with "Johan Sebastian Bach", then, in terms of the figure, the first phrase, *qua* sentence of English, would be N_1; it would "notate" or "express" the mentalese structure N_1, and the person who lived in the seventeenth and eighteenth centuries would be the referent D_1. Similarly, my reply would be N_2, the mentalese fragment that I thereby express would be S_2, and D_2 would again be the long-dead composer. I.e., in this case D_1 and D_2 would be identical.

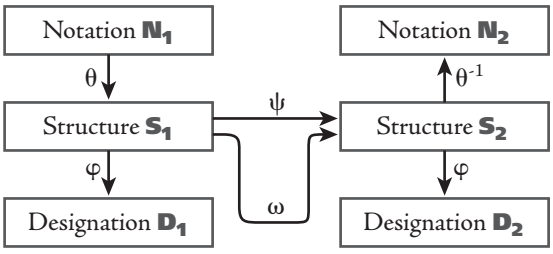

Figure 7

N_1, S_1, D_1, N_2, S_2, and D_2, in other words, need not necessarily all be distinct; in a variety of different circumstances two or more of them may be one and the same entity. I will examine cases, for example, of self-referential designators, where S_1 and D_1 are the same object. Similarly, if, on hearing the phrase "the A53

13. Fodor (1980).

pseudonym of Samuel Clemens," I reply "Mark Twain", then D_1 and N_2 are identical. By far the most common situation, however, will be as in the Bach example, where D_1 and D_2 are the same entity—a circumstance in which I will say that the function ψ is **designation-preserving**. As we will see in the next section, the α-reduction and β-reduction of the λ-calculus, and the derivability relationship (⊢) of logic, are both designation-preserving relationships. Similarly, the 2Lisp and 3Lisp processors I present will be designation-preserving, whereas 1Lisp's and Scheme's evaluation protocols, as we have already indicated, are not.

In the terms of this figure, the argument I will present in [dissertation] chapter 3 will run roughly as follows. First I will review both logic systems and the λ-calculus, to illustrate the general properties of the φ and ψ employed in those formalisms, for comparison. Next I will shift towards computational systems, beginning with Prolog, since it has evident connections to both declarative and procedural traditions. Finally I will take up Lisp. I will argue that it is not only coherent, but in fact natural, to define a declarative φ for Lisp, as well as a procedural ψ. I will also sketch some of the mathematical characterisation of these two interpretation functions. It will be clear that though similar in certain ways, they are nonetheless crucially distinct. In particular, I will be able to show that 1Lisp's ψ (EVAL) obeys the following equation. I will say that any system that satisfies this equation has the **evaluation property**, and the statement that, for example, the equation holds of 1Lisp the **evaluation theorem**. (The formulation used here is simplified for perspicuity, ignoring contextual relativisation; $\mathcal{S}$ is the set of structural field elements.)

$$\forall s \in \mathcal{S}\ [\ \text{if } \varphi(s) \in \mathcal{S}\ \text{ then } \psi(s)=\varphi(s) \quad \text{else } \varphi(\psi(s))=\varphi(s)\ ] \qquad [5]$$

1Lisp's evaluator, in other words, **de-references** *just those struc-*

tures whose referents lie within the structural field, and is designation-preserving otherwise. Where it can, in other words, 1Lisp's ψ (i.e, its processor) implements φ; when it cannot, ψ is φ*-preserving,* although what it does do with its argument in this case has yet to be explained (saying that it preserves φ is too easy: the identity function preserves designation as well, but EVAL is not the identity function).

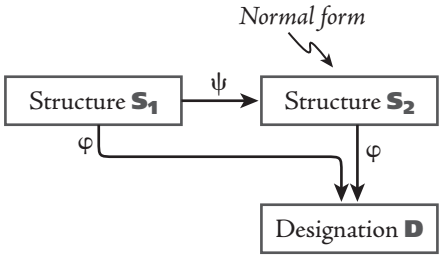

Figure 8

The behaviour described in [5] is unfortunate, in part because the question of whether $\varphi(s) \in \mathcal{S}$ is not in general decidable, and therefore even if one knows of two expressions s_1 and s_2 that s_2 is $\psi(s_1)$, one still does not necessarily know the relationships between $\varphi(s_1)$ and $\varphi(s_2)$. More seriously, it makes the explicit use of meta-structural facilities extraordinarily awkward, thus defeating attempts to engender reflection. I will argue instead for a dialect described by the following alternative (again in skeletal form):

$$\forall s \in \mathcal{S}\ [[\phi(\psi(s)) = \phi(s)] \wedge [\text{NORMAL-FORM}(\psi(s))]] \quad [6]$$

When I prove it for 2Lisp, I will call this equation the **normalisation theorem**; I will say that any system satisfying it has the **normalisation property**. Diagrammatically. the circumstance it describes is pictured in figure 8. Such a ψ, in other words, is *always* φ-preserving. In addition, it relies on a notion of normal-formedness, which we will have to define.

In the λ-calculus, ψ(s) would *definitionally* be in normal-form, since in that calculus normal-formedness is *defined* in terms of the non-applicability of any further β-reductions. As I will argue in more detail in [dissertation] chapter 3, this makes the notion less than ideally useful: in designing 2Lisp and 3Lisp. In contrast, therefore, I will define normal-formed-

ness in terms of the following three (provably independent) properties:

1. Normal-form designators must be **context-independent**, in the sense of having the same declarative and procedural import independent of their context of use;
2. They must also be **side-effect free**, implying that any (further) procedural treatment of them will have no affect on the structural field or state of the processor;
3. They must be **stable**, meaning that they normalise to themselves in all contexts.

It will then require a proof that all 2Lisp and 3Lisp results (all expressions $\psi(s)$) are in normal-form. In addition, from the third (stability) property, plus this proof that ψ's range includes only normal-form expressions, it will be possible to show that ψ is *idempotent*, as was suggested earlier ($\psi=\psi\circ\psi$—i.e., $\forall s\ \psi(s)=\psi(\psi(s))$)—a property of 2Lisp and 3Lisp that will ultimately be shown to have substantial practical benefits.

There is another property of normal-form designators in 2Lisp and 3Lisp, beyond the three requirements just listed, which follows from the category alignment mandate. In designing those dialects I will insist that the *structural category* of each normal form designator be determinable from *the type of object designated*, independent of the structural type of the original designator, and independent as well of any of the machinery involved in implementing ψ (this is in distinction to the received notion of normal form employed in the λ-calculus, as will be examined in a moment). For example, I will be able to demonstrate that any term that designates a number will be taken by ψ into a numeral, since numerals will be defined as the normal-form designators of numbers. In other words. from just the designation of a structure s the *structural category* of $\psi(s)$ will be predictable, independent of the form of s itself (although the *token identity* of $\psi(s)$ cannot

be predicted on such information alone, since normal-form designators are not necessarily unique or canonical). This category result, however, will also need to be proved: I call it the **semantical type theorem**.

That normal form designators cannot be canonical arises, of course, from computability considerations: one cannot decide in general whether two expressions designate the same function, and therefore if normal-form function designators were required to be unique, it would follow that expressions that designated functions could not necessarily be normalised. Instead of pursuing that approach, however, which I would view as unhelpful, I will instead adopt a non-unique notion of normal-form function designator, which still satisfies the three requirements specified above; such a designator will by definition be called a **closure**. All well-formed function-designating expressions, on this scheme, will succumb to a standard normalisation.

Some 2Lisp (and 3Lisp) examples will illustrate all of these points. I assume that the numbers are included in the semantical domain, a syntactic [i.e., structural] class of **numerals** are taken to be normal-form number designators. The numerals are canonical (one per number), and as usual are side-effect A58 free and context-independent; thus they satisfy the requirements on normal-formedness. The semantical type theorem says that any term that designates a number will normalise to a numeral: thus if X designates five and Y designates six, and if + designates the addition function, then we know (can prove) that (+ X Y) designates eleven and will normalise to the numeral 11. Similarly, there are two boolean constants $T and $F that are normal-form designators of Truth and Falsity, respectively, and a canonical set of rigid structure designators called **handles** that are normal-form designators of all s-expressions (including themselves). And so on: closures are normal-form

function designators, as mentioned above; I will also specify normal-form designators for sequences and other types of mathematical objects included in the semantic domain.

I have diverted the discussion away from general semantics, onto the particulars of 2Lisp and 3Lisp, in order to illustrate how the semantical reconstruction I endorse impinges on language design. However, it is important to recognise that the behaviour mandated by [6] is not *new:* this is how all standard semantical treatments of the λ-calculus proceed, and the designation-preserving aspect of it is approximately true of the inference procedures in logical systems as well, as we will see in detail in [dissertation] chapter 3. Neither the λ-calculus reduction protocols, in other words, nor any of the typical inference rules one encounters in mathematical or philosophical logics, *de-reference* the expressions over which they are defined. In fact it is hard to imagine *defending* equation [5]. Rather, it seems reasonable to speculate that because Lisp includes its syntactic domain within the semantic domain—i.e., because Lisp has QUOTE as a primitive "operation"—a semantic inelegance was inadvertently introduced into the design of the language that has never been corrected. If this is right, then the proposed rationalisation of Lisp can be understood as an attempt to *regain the semantical clarity* of predicate logic and the λ-calculus, achieved in part by connecting the language of the computational calculi with the language in which prior linguistic systems have been studied. A59

It is this regained coherence that I am claiming is a necessary prerequisite to a coherent treatment of reflection.

One final comment The consonance of [6] with standard semantical treatments of the λ-calculus, and the comments just made about Lisp's inclusion of QUOTE, suggest that one way to view the present project is as a semantical analysis of a variant of the λ-calculus with quotation. In the Lisp dialects

I consider, I will retain sufficient machinery to handle side effects, but it is of course always possible to remove such facilities from a calculus. Similarly, we could remove the numerals and atomic function designators (i.e., the ability to name composite expressions as unities). What would emerge would be A60
a semantics for a deviant λ-calculus with some operator like QUOTE included as a primitive syntactic construct—a semantics for a *meta-structural* extension of the already *higher-order* λ-calculus. I will not pursue this line of attack further in this dissertation, but, once the mathematical analysis of 2Lisp is in place, such an analysis should emerge as a straightforward corollary.

1d·iii Recursive and Compositional Formulations

The previous sections have briefly suggested goals for the semantical account to be developed, but they say nothing about how those goals can be reached. In [dissertation] chapter 3, where the reconstruction of semantics is laid out, I will of course pursue this latter question in detail, but I can summarise some of its results here.

Beginning very simply, standard approaches suffice. For example, I begin with *declarative import* (φ), and initially posit the designation of each primitive object type (saying for instance that the numerals designate the numbers, and that the primitively recognised closures designate a certain set of functions, and so forth), and then specify recursive rules that show how the designation of each composite expression emerges from the designation of its ingredients. Similarly, in parallel fashion I specify the *procedural consequence* (ψ) of each primitive type (saying in particular that the numerals and booleans are *self-evaluating,* that atoms evaluate to their bindings, and so forth), and then once again specify recursive rules showing how the value or *result* of a composite expression is formed from the results of processing its constituents.

If we were considering only purely extensional, side-effect free, functional languages, the story might end there. However a61 a variety of complications will demand resolution, of which two may be mentioned here. First, none of the Lisps that I will consider are purely extensional: there are intensional constructs of various sorts (QUOTE, for example, and even LAMBDA, which I will view as a standard intensional procedure, rather than as a syntactic mark). The hyper-intensional QUOTE operator is not in itself difficult to deal with, although I will also consider questions about the less fine-grained intensionality a62 manifested by a statically-scoped LAMBDA. As in any system, the ability to deal with intensional constructs requires a reformulation of the semantics of all expressions—i.e., requires recasting the semantics of extensional procedures as well, in appropriate ways. This is a minor complexity, but no particular difficulty emerges.

The second complication has to do with side-effects and contexts. All standard model-theoretic techniques of course allow for the general fact that the semantical import of a term may depend in part of on the context in which it is used (variables are the classic simple example). However, side-effects—which are part of the total *procedural consequence* of an expression, impinge on the appropriate context *for declarative purposes* as well as well as for procedural ones. For example, in a63 a context in which X is bound to the numeral 3 and Y is bound to the numeral 4, it is straightforward to say that the term (+ 3 Y) designates the number seven, and returns the numeral 7. However consider the semantics of the following more complex expression (this is standard Lisp) when evaluated in the same context:

(+ 3 (PROG (SETQ Y 14) Y)) [7]

It would be hopeless—to say nothing of false—to have the formulation of declarative import ignore procedural conse-

quence, and claim that [7] designates seven, even though it patently returns the numeral 17 (although I am under no obligation to make the declarative and procedural stories cohere in anything like an astethic sense—in fact I will reject 1-Lisp exactly because they do *not* cohere in any way that I can accept). On the other hand, to *include* the procedural effect of the SETQ within the specification of φ would seem to violate the ground intuition arguing that the designation of this term, and the structure to which it evaluates, are different.

The approach I will ultimately adopt is one in which I define what I call a **general significance function** Σ which embodies both declarative import (designation), local procedural consequence (what an expression "evaluates to," to use 1Lisp jargon), and full procedural consequence (the complete contextual effects of an expression, including side-effects to the environment, modifications to the structural field, and so forth). Only the total significance of the dialects I define will be strictly *compositional;* the components of that total significance, such as the designation, will be *recursively specified* in terms of the designation of the constituents, relativised to the total context of use specified by the encompassing general significance function. In this way I will be able to formulate precisely the intuition that the expression given in [7] designates seventeen, as well as returning the corresponding numeral 17.

Lest it seem that by handling these complexities we have lost any incisive power in the approach, I should note that it is not always the case that the processing of a term results in the obvious (i.e., normal-form) designator of its referent For example, I will prove that, in traditional Lisps, the expression

(CAR '(A B C)) [8]

both designates *and* returns the atom A. Just from the contrast between these two examples ([7]and [8]) it is clear that traditional Lisp processing and Lisp designation do not track each other in any trivially systematic way.

Although this approach will be shown successful, I will ultimately abandon the strategy of characterising the full semantics of standard Lisp (as exemplified in my ILisp dialect), since the confusion about the semantic import of evaluation will in the end make it virtually impossible to say anything coherent about designation. This, after all, is my goal: to *judge* ILisp, not merely to *characterise* it. By the time I wrap up its semantical analysis, I will have shown not only *that* Lisp is confusing, but also (in detail) *why* it is confusing—giving us adequate preparation to design a dialect that corrects its errors.

1d·iv The Role of a Declarative Semantics

One brief final point about this double semantics.

It should be clear that it is impossible to specify a normalising processor without a pre-computational, non-procedural theory of semantics. If you do not have an account of what structures mean, *independent of* and how they are treated by the processor, there is no way to say anything substantial about the semantical import of the function that the processor computes. On the standard approach, for example, it is impossible to say that the processor is *correct*, or *semantically coherent*, or *semantically incoherent*, or any such thing; it would merely be what it is. Given some account of what it does, one can compare this to *other* accounts: thus it would for example be possible to prove that a *specification* of it was correct, or that an *implementation* of it was correct, or that it had certain other independently definable properties (such as that it always terminated, that it used certain resources in certain fashion, etc.). In addition, *given* such an account, one could prove properties of programs written in the resulting language—thus, from a mathematical specification of the processor of ALGOL, plus the listing of an ALGOL program, it might be possible to prove that that program met some specification (such as that it sorted its input, or whatever). But all of these things are compatible

A64 A65

with the system being a purely mechanical system—such as a device that sorted apples into different bins, or for that matter was a car. However none of these questions are the question I am trying to answer here—namely: *what is the semantical character of the processor itself?*

In the particular case I am considering, I will be able to specify the semantical import of the function computed by Lisp's evaluation regimen (i.e., by EVAL—this is content of the evaluation theorem), but only by first laying out both declarative and procedural theories of Lisp. Again, I will be able to design 2Lisp only with reference to this pre-computational theory of declarative semantics. It is a simple point, which I am perhaps repeating too often, but it is important to make clear how the semantical reconstruction I am endorsing is a *prerequisite* to the design of 2Lisp and 3Lisp, not a post-facto method of analysing them.

1e Procedural Reflection

Now that we have assembled a minimal vocabulary with which to talk about computational processes and matters of semantics, it is possible to sketch the architecture of reflection that I will present in the final chapter of the dissertation.

I will start rather abstractly, with the general sense of reflection sketched in §1·b, and then make use of both the Knowledge Representation Hypothesis and the Reflection Hypothesis to define a more restricted goal. Next, I will employ the characterisations of [serially] reduced computational processes and of computational semantics to narrow this goal even further. At each step in this progressive focusing process, it will become increasingly clear what would be be involved in actually constructing an authentically reflective computational language. By the end of this section I will be able to suggest the particular structure that, in [dissertation] chapter 5, will be embodied in the 3Lisp design.

1e·i A First Sketch

Begin very simply. At the outset, I characterised reflection in terms of a process shifting between a pattern of reasoning about some subject matter, world, or task domain, to reasoning reflectively about its thoughts and actions in that world. I said in the Knowledge Representation Hypothesis that the only current candidate architecture for a process that reasons *at all* (even derivatively) is one constituted in terms of an interior process manipulating representations of the appropriate knowledge of that domain. We can see in terms of the process reduction model of computation a little more clearly what this means. For the process I called Chequers to reason about the world of finance, I suggested that it be [serially] composed of an ingredient process P manipulating a structural field S consisting of representations of cheque books, credit and debit entices, currency exchange rates, and so forth. Thus we were led to the image depicted in figure 4 (reproduced here as figure 9).

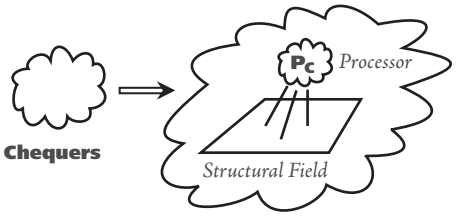

Figure 9

Next, I said in the Reflection Hypothesis that the only suggestion we have as to how to make Chequers reflective is this: as well as constructing process P_c to "deal with" (that is: manipulate symbols denoting) these various financial records, we could also construct process Q to deal with P *and the structural field that* P_c *manipulates*. Thus Q might specify what to do when P_c failed or encountered an unexpected situation, based on what parts of P_c had worked correctly and what state P_c was in when the failure occurred, and so on. Alternatively, Q might describe or generate parts of P_c that had not been fully or adequately specified. Finally, Q might bring into existence a more complex interpretation process for P_c, or one particularised

to suit specific circumstances—thereby engendering something we might want to call P_c'. In general, whereas the world of P_c—the domain that P_c models, simulates, reasons about, onto which the declarative interpretation function φ maps its ingredient symbols—is the world of finance, the corresponding world of Q is the world of the process P_c and the structural field it computes over.

I have spoken as if Q were a *different* process from P_c, but whether it is really different from P_c, or whether it is P_c in a different guise, or P_c at a different time, is a question I will defer for a while (in part because I have said nothing about individuation criteria on processes). All that matters for the moment is that there be *some* process that does what I have said that Q must do.

What is required, in order for Q to reason about P_c? Because Q, like all the processes we are considering, is assumed to be [serially] composed, what is needed is what is always needed: *structural representations of the relevant facts about* P_c. What would such representations be like? First, they must be expressions (statements or symbols), formulated with respect to some theory, describing or representing the state of process P_c (we can begin to see how the *theory-relative* mandate on reflection from §1·b is making itself evident). Second, in order to actually describe P_c, they must be *causally connected* to P_c in some appropriate way (another of the general requirements). Thus we are considering a situation such as that depicted in figure 10, where the field (or field fragment) S_P contains these causally connected structural descriptions.

Figure 10

Figure 10 is of course incomplete, in that it does not sug-

gest how s_p should relate to p_c (answering this question is our current quest). Note however that reflection must be able to recurse, implying the additional possibility of something like the image depicted in figure 11.

Where might an encodable procedural theory come from? There are two possible sources: in the semantical reconstruction to be undertaken presently (before 3Lisp is designed) I will have presented a full theory of the (non-reflective versions of the) dialects under development; this is one candidate source for an appropriate theory. But given that for the moment we are considering only procedural reflection, the simpler procedural component will suffice (in contrast to the general case, where we would need to encode the full theory of computational significance).

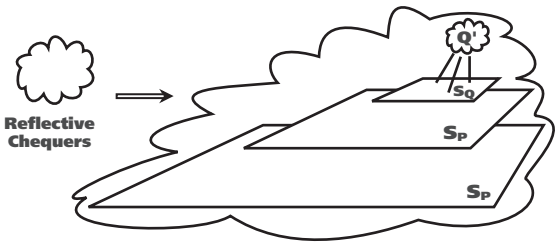

Figure 11

The second source of a theoretical account, quite similar in structure but even closer to the one we will adopt, is what we will call the **metacircular processor**, which is worth a brief examination.

1e·ii Metacircular Processors

In any computational formalism in which programs are accessible as first class structural fragments, it is possible to construct what are commonly known as *metacircular interpreters:* "meta" because they operate on (and therefore terms within them designate) other formal structures, and "circular" because they do not constitute a definition of the processor, for two reasons: (i) they have to be run by that processor in order to yield any sort of behaviour (since they are *programs*, not

processors, strictly speaking); and (ii) the behaviour they would thereby engender can be known only if one knows beforehand what the processor does. Nonetheless, such processors are often pedagogically illuminating, and they will play a critical role in our development of the 3Lisp reflective model. In line with my general strategy of reserving the word "interpret" for the semantical interpretation function. I will henceforth call such processors **metacircular processors**.

In the presentation of 1Lisp and 2Lisp I will construct metacircular processors (MCPs); the 2Lisp version is presented in figure 12, on the next page (details will be explained in [dissertation] chapter 4; at the moment I mean only to illustrate the general structure of this code). The basic idea is that if this code were processed by the primitive 2Lisp processor. the process that would thereby be engendered would be behaviourally equivalent to that of the primitive processor itself. In other words, if we were mathematically to take processes as functions from structure onto behaviour, and if we name the processor presented in figure 12 MCP_{2L}, and the primitive 2Lisp processor P_{2L}, then if we taken '$\simeq$' to mean behaviourally equivalent, then we should be able to prove the following, in some appropriate sense (this is the sort of proof of correctness one finds in for example Gordon[14]):

$$\text{P}_{2L}(\text{MCP}_{2L}) \simeq \text{P}_{2L} \qquad [9]$$

It should be recognised that the equivalence spoken of here is a global equivalence; by and large the primitive processor, and the processor resulting from the explicit running of the MCP, cannot be arbitrarily mixed (as already mentioned, and as a more detailed discussion in [dissertation] chapter 5 will formalise). For example, if a variable is bound by the underlying processor P_{2L} it will not be able to be looked up by the metacircular code. Similarly, if the metacircular processor encounters a control structure primitive, such as a THROW or a QUIT, it will

14. Gordon (1973 and 1975).

not cause the metacircular processor itself to exit prematurely, or to terminate. The point, rather, is that if an entire computation is mediated by the explicit processing of the MCP, then the results will be the same as if that entire computation had been carried out directly.

We can merge these results about MCPs in general with the diagram in figure 9 as follows: if we replaced P in the figure

```
(define NORMALISE
   (lambda expr [exp env cont]
      (cond [(normal exp) (cont exp)]
            [(atom exp) (cont (binding exp env))]
            [(rail exp) (normalise-rail exp env cont)]
            [(pair exp) (reduce (car exp) (cdr exp) env cont)])))

(define REDUCE
   (lambda expr [proc args env cont]
      (normalise proc env
         (lambda expr [proc!]
            (selectq (procedure-type proc!)
               [impr (if (primitive proc!)
                         (reduce-impr proc! args env cont)
                         (expand-closure proc! args cont))]
               [expr (normalise args env
                         (lambda expr [args!]
                            (if (primitive proc!)
                               (reduce-expr proc! args! env cont)
                               (expand-closure proc! args! cont))))]
               [macro (expand-closure proc! args
                            (lambda expr [result]
                               (normalise result env cont)))])))))

(define EXPAND-CLOSURE
   (lambda expr [closure args cont]
      (normalise (body closure)
                 (bind (pattern closure) args (env closure))
                 cont)))
```

Figure 12

with a process that resulted from P processing the metacircular processor MCP (for the appropriate language—in this case assumed to be Fortran), we would still correctly engender the behaviour of Chequers, as depicted in figure 13. Furthermore, this replacement could also recurse, as shown in figure 14, on the next page. Admittedly, under the standard interpretation, each such replacement would involve a dramatic decrease in efficiency, but the important point is that, modulo those temporal issues, the resulting behaviour would in some sense still be correct.

1e·iii Procedural Reflective Models

We are now in a position to unify the suggestion made at the end of §1·e·ii, on having Q reflect upwards, with the insights embodied in the MCPs described in the previous section, to define what I will call the **procedural reflective model**. The fundamental insight arises from the eminent similarity between figures 10 and 11, on the one hand, compared with figures 13 and 14, on the other. These diagrams do not represent exactly the same situation, but the approach will be to converge on a unification of the two.

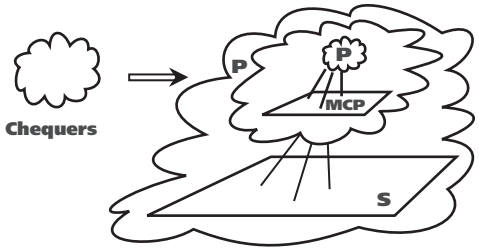

Figure 13

I said earlier that in order to satisfy the requirements on the Q of §1·e·ii we would need to provide a causally connected structural encoding of a procedural theory of our dialect (Lisp in this case) within the accessible structural field. In the immediately preceding section we have seen something that is *approximately* such an encoding: the metacircular processor. However—and here I refer back to the six properties of reflection set out in

§1·b·iii—in the normal course of events the MCP lacks the appropriate causal access to the state of P: whereas any possible state of Q *could* be procedurally encoded in terms of the metacircular process (i.e., given any account of the state of P we could retroactively construct appropriate arguments for the various procedures in the metacircular processor so that if that metacircular processor were run with those arguments it would mimic P in the given state), in the normal course of events the state of P will *not be so encoded*.

This similarity, however, does suggest the form of the solution.

Suppose that P were *never* run directly, but were *always* run in virtue of the explicit mediation of the metacircular processor—as, for example, in figure 13 and 14. Then at any point in the course of the computation, if that running of one level of the MCP were *interrupted*, and the arguments being passed around were used by some *other procedures*, they would be given just the needed information: correct causally connected representations of the state of the process P prior to the point of reflection. The MCP would of course have to be modified in order to support such an interruption; the point however is that the MCP is already trafficking in the requisite causally connected representations.

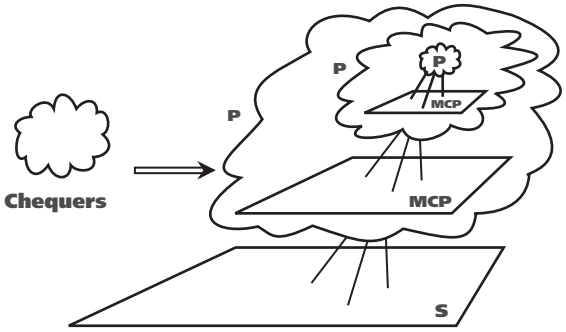

Figure 14

There are however evident problems with this approach. First, if P were always run through the mediation of the

metacircular processor MCP, P would as a result almost surely be unnecessarily inefficient. Second, as so far stated the proposal seems to deal with only one level of reflection. What if the code that was given these structural encodings of P's state was itself to reflect? This query suggests that providing a general mechanism for reflection would generate an infinite regress: not only should the MCP be used to run the base ("level 0") programs, but the MCP should be used to run the level 1 MCP. And so on. That is: *all* of an infinite number of MCPs should be run by yet further MCPs, ad infinitum.

Setting aside the obvious vicious regress for a moment, note that this seems otherwise to be a reasonable suggestion. The *potentially* infinite (i.e., indefinite) set of reflecting processes Q are almost indistinguishable in basic structure from the infinite tower of MCPs that would result. Furthermore the MCPs would contain just the correct structurally encoded descriptions of processor state. We would still need to modify the whole set of MCPs, so that an appropriate interruption or reflective act could make use of the tower of processes, but it is nevertheless evident that, to a first degree of approximation, this proposal has the proper character.

The fundamental "trick" of 3Lisp (i.e., of the model of procedural reflection being proposed) hinges on the fact that, it turns out, we can effectively *posit*, as a stipulative but extremely useful fiction, that *the primitive reflective processor is engendered by an infinite number of recursive instances of the* MCP, *each running a version one level below*. That is: 3Lisp will be defined to be isomorphic to that infinite limit. This turns out to be legitimate—i.e., the implied infinite regress is not after all problematic—since only a finite amount of information is encoded in it; at all but a finite number of the bottom levels, each MCP will merely be running a copy of the MCP. Because we, as the language designers, know exactly how the language runs, and

because we also know what the MCP is like, we can provide this infinite numbers of levels, to use current jargon, *purely virtually*. As I will explain in detail in [dissertation] chapter 5, such a virtual simulation turns out to be perfectly well-defined.

Once the changes are made to support appropriate interruption and resumption at any arbitrary level, it is no longer appropriate to call the processor a *metacircular* processor, since it becomes inextricably woven into the fundamental architecture of the language (as will be explained in detail in [dissertation] chapter 5). This is why, as suggested above, I call it a *reflective processor*. Nonetheless its genealogical roots in the abstract idea of an infinite tower of metacircular processor should be clear.

To provide a little bit of concrete grounding for this suggestion, I will explain just briefly the "interruption adjustment" we will make in order to allow this architecture to be used.

3Lisp supports what I will call **reflective procedures**—procedures that, when invoked, are run not at the level at which the invocation occurred, but one level higher in the **reflective hierarchy.** They are given, as arguments, *those structures that would have been passed around in the reflective processor, had it always been running explicitly*. The code for the resulting 3Lisp reflective processor program is given in figure 15 (next page) in part so that it may be compared with the (very similar) 2Lisp meta-circular processor code given earlier in figure 12. The most important difference lies on a single line, underlined here for emphasis.

What is important about the underlined line (line 18) is this: when a redex (application) is encountered whose CAR normalises to a reflective as opposed to standard procedure (the standard ones are called "SIMPLE" within this dialect), the corresponding function, designated by the term ↓(de-reflect proc!), is run *at the level of the reflective processor,* rather than

by the processor. In other words the inclusion of this single underlined line unleashes the full infinite reflective hierarchy.

```
1  (define READ-NORMALISE-PRINT
2  ... (lambda simple [level env stream]
3  .....(normalise (prompt&read level stream) env
4  ..... (lambda simple [result]                                  Ç-REPLY
5  .....(block (prompt&reply result level stream)
6  ..... (read-normalise-print level env stream))))))

7  (define NORMALISE
8  ... (lambda simple [struc env cont]
9  .....(cond [(normal struc) (cont struc)]
10 .....[(atom struc) (cont (binding struc env))]
11 .....[(rail struc) (normalise-rail struc env cont)]
12 .....[(pair struc) (reduce (car struc) (cdr struc) env cont)]))

13 (define REDUCE
14 ... (lambda simple [proc args env cont]
15 .....(normalise proc env
16 ..... (lambda simple [proc!]                                  Ç-PROC!
17 .....(if (reflective proc!)
18 ..... (↓(de-reflect proc!) args env cont)
19 ..... (normalise args env
20 ..... (lambda simple [args!]                                  ; C-ARGS!
21 ..... (if (primitive proc!)
22 .....(cont ↑(↓proc! . ↓args!))
23 .....(normalise (body proc!)
24 .....(bind (pattern proc!) args! (environment proc!))
25 .....cont))))))))

26 (define NORMALISE-RAIL
27 ... (lambda simple [rail env cont]
28 .....(if (empty rail)
29 ..... (cont (rcons))
30 ..... (normalise (1st rail) env
31 .....(lambda simple [first!]                                   ; C-FIRST!
32 .....(normalise-rail (rest rail) env
33 .....(lambda simple [rest!]                                   ; C-REST!
34 ..... (cont (prep first! rest!)))))))))
```

Figure 15 — The 3Lisp Reflective Processor

Coping with that hierarchy will occupy part of [dissertation] chapter 5, where I explain this all in much more depth (including why the resulting virtual machine is in fact finite, and how it can be implemented). Just this much of an introduction, however, should convey, if only a glimpse of how reflection is possible.

1e·iv Two Views of Reflection

The reader will have noted a tension between two ways in which I have characterised the form of reflection we are aiming at. On the one hand I have sometimes written as if there were a primitive and noticeable **reflective act**, which causes the processor to **shift levels** rather markedly (this is the explanation that best coheres with some of our pre-theoretic intuitions about reflective human thinking). On the other hand, I have also just written of an infinite number of levels of re1ective processors, each essentially implementing the one below—a story according to which it is not coherent either to ask at which level 9 is running, or to ask how many reflective levels are running. On this "infinite tower" account, there is a strong some sense in which *all levels are running at once*, in exactly the same sense that both the Lisp processor inside your Lisp-based editor, and your editor itself, and the machine language code that underpins the implementation of Lisp, are all running at once, when you use the editor. It is of course not as if Lisp, the editor, and the machine language are running simultaneously in the sense of *side-by-side* or *independently*. This is not a parallel computing scheme being described. Rather, in each case one, being "interior" to the other, supplies the anima or agency of the outer one (machine language processor animating the Lisp processor, which in turn animates the editor). It is just this sense in which the higher levels in the 3Lisp reflective hierarchy are always running: each of them is in some sense *within* (interior to) the processor at the level below it, in such a way that it thereby engenders it.

Call the account that views reflection as a case of a single locus of agency stepping between levels the **level-shifting** view. And call the other view that of an **infinite tower**. I will not take a principled view on which is correct; for certain A67 purposes one is simpler, for others the other. What matters most is to recognise their behavioural equivalence—or to put it in a little more detail: the fundamental architectural thesis underlying not only 3Lisp in particular but the general model of procedural reflection being proposed here is that *embracing the limiting behaviour of the tower view* is an appropriate ideal in terms of which to design, understand, and implement the level-shifting view.

Though perhaps more initially intuitive, the level-shifting account turns out to be more complex than the tower view. To illustrate it, consider the following account of what is involved in constructing a reflective dialect—in part by way of review, but also in order to suggest how it is that a practical reflective dialect could be finitely constructed.

1. As I have repeatedly said, in order to design a reflective language one must provide a complete theory of the given calculus expressed in its own language. I call this the **reflective processor**—it is required on both accounts.
2. You must arrange things so that, when the process reflects—i.e., when, on the level-shifting view, the locus of control shifts "upwards"—all of the structures used by the reflective processor (the formal structures designating the theoretical entities posited by the theory) are available for inspection and manipulation. In any particular case, these to-be-provided structures must correctly encode *the state that the processor was in prior to the reflective level-shift, assuming that it had been running all the while* (this is where the tower view provides

structure and substance—fills in the technical details—for the level shifting view).

3. You must also ensure, when the (level-shifting) process comes to the point of "shifting down" again, that base-level processing is resumed *in accordance with the facts encoded in the structures being passed around at the immediately higher reflective level*.

As a minimal case, take a situation where the user process shifts upwards, but does nothing; and then shifts down again. At the point of shifting up, the situation should merely be one where the processor would process the reflective processor code explicitly, as if it had been doing so all along. At the point of shifting down, it would take up running the base-level code directly (i.e., non-reflectively), again *as if it had been doing that all along*, but also (of course it must be proved that these are equivalent) exactly in accord with the state of the structures being passed around in the reflective processor code at the point of down-shifting. Such a situation, in fact, is *so* simple that it could not be distinguished (except perhaps in terms of elapsed time) from pure non-reflective interpretation.

The situation would get more complex, however, as soon as the user is given any power. Two provisions in particular are crucial.

First, the whole purpose of a reflective dialect is to allow the user to have his or her own programs run along with, or in place of, or between the steps of, the reflective processor. One must in other words provide an abstract machine with the ability for the programmer to insert code—in convenient ways and at convenient times—at any level of the reflective hierarchy. Suppose, for example, we were to wish to have a particular λ-expression closed only in the dynamic environment of its use, rather than in the lexical environment of its definition (i.e., suppose we were to want "dynamic scoping" for

a given λ-expression, even though lexical scoping is the system default). Needless to say, the reflective processor contains code that performs the requisite operations needed to implement the default behaviour for lexical closures. Given that the programmer can assume that, upon reflection, the reflective processor code is being explicitly processed, he or she can supply, for the λ-expression in question, an appropriate *alternate piece of code* for the reflective process, in which different actions are taken so as to provide the special λ-expression with dynamic scoping behaviour. By simply inserting this code into the correct level, (s)he can use variables bound by the reflective model in order to fit gracefully into the overall processing regimen. Appropriate hooks and protocols for such insertion, of course, must be provided, but they need be provided only once. Furthermore, the reflective processor code (i.e., reflective model) will contain code showing how this hook is treated.

All of these requirements are met by the underlined line 18 in the reflective processor program of figure 15. That line indicates how the user code will be inserted, what context it will run it, what variables will be bound to what structures containing what information, etc.

Second, as well as providing for the arbitrary interpretation of special programs at the reflective level, the language designer must also enable the user to *modify* the explicitly available structures provided in the reflective model. Though this ability is easier to design than the former, its correct implementation is trickier. An example will make this clear. As already indicated, the 3Lisp reflective processor deals explicitly with both environment and continuation structures. Upon reflecting, user programs can at will access these structures that, at the base level, are purely implicit. Suppose that a user writes reflective code that does two things. First, it modifies the environment structure being passed around at the first reflective level (e.g., suppose it changes the binding of a variable bound

by some procedure that is running "somewhere up the stack," in the way that might be provided by a typically debugging package). Second, it changes the continuation structure (designating the continuation function) so as to cause some procedure that is currently running to, upon its return, bypass its immediate caller, and instead return its result to the procedure that called that procedure. Then, once this user code has effected these two changes, it "returns"—which is to say, it "drops back down" to other base-level code, and no longer runs at the reflective level.

I said above that, upon this kind of semantic or reflective descent, the base-level program will again be processed "directly." But of course it must be processed in such a way as to honour the changes indicated by these modified structures—not in the way that it would have proceded, prior to the reflection. The user's reflective modifications, in other words, must *matter*—must be *noticed*. This is the (downwards direction of) the *causal connection* aspect that is so crucial to true reflection.

1e·v General Comments

The details of the proposed architecture have emerged from detailed considerations of process reduction, computational semantics, and meta-circular processing. It is interesting to draw back and to see the extent to which the global properties of the resulting architecture match our pre-theoretic intuitions about reflection.

First, it is simple to see that the proposed architecture honours all six requirements laid out in §1.b.iii:

1. It is causally connected;
2. It is theory-relative;
3. It involves an incremental "stepping back," rather than a full (and potentially vicious) instantaneous "reflexion";
4. Finer-grained control is provided over the processing of lower level structures;

5. It is only partially detached (3Lisp reflective procedures are still in and part of 3Lisp; they are still animated by the same fundamental agency, since if one level stops processing the reflective model, or some analogue of it, all the processors "below" it cease to exist): and
6. The reflective powers of 3Lisp are primitively provided.

Thus in this sense at least it is fair to count the architecture a success.

Other questions—such as about the locus of self, the concern as to whether the potential to reflect requires that one always participate in the world indirectly rather than directly, and so forth—turn out to be about as difficult to answer for 3Lisp as they are to answer in the case of human reflection. In particular, the solution I have proposed does not answer the question I posed earlier, about the identity of the reflected processor: is it P that reflects, or is it another process Q that reflects on P? The "reflected process" is neither quite the same process, nor quite a different process; it is in some ways as different as an *interior* process, except that since it shares the same structural field it is not as different as an implementing process. No more informative answer will be forthcoming until we define individuation criteria on processes much more precisely—and, perhaps more strikingly, there seems no particular reason to answer the question one way or another. It is tempting (if dangerous) to speculate that the reason for these difficulties in the human case is exactly why they do not have answers in the case of 3Lisp: they are not, in some sense, "real" questions. But it is premature to draw this kind of parallel; our present task is merely to clarify the structure of proposed solution.

1f Lisp as an Explanatory Vehicle

There are any number of reasons why it is important to work with a specific programming language, rather than abstractly

and in general (for pedagogical accessibility, as a repository for emergent results, as an example to test proposed technical solutions, and so forth). Furthermore, commonsense considerations suggest that a familiar dialect, rather than a totally new formalism, would better suit our purposes. On the other hand there are no current languages that are categorically and semantically rationalised in the way that the proposed theory of reflection demands; according to the mandate that "reflection is intelligibly implementable only on a semantically clarified basis," it is not an option to endow any extant system with reflective capabilities without first subjecting it to substantial modification. It would be possible to present a new system embodying all the necessary modifications and features, but it would be difficult for the reader to sort out which architectural features were due to what concern. In this dissertation, therefore, I have adopted the strategy of presenting a reflective calculus in two steps: first, by modifying an existing language to conform to the outlined *semantical* mandates (2Lisp); and second, by extending the resulting rationalised language with *reflective* capabilities (3Lisp).

Once this overall plan has been agreed, the question arises as to what language should be used as a basis for this two-stage development Since my present concern is with *procedural* rather than with *general* reflection, the relevant class of potential languages includes essentially all programming languages, but excludes exemplars of the declarative tradition: logic, the λ-calculus, specification and representation languages, and so forth. Furthermore, we need a programming language—a A68
procedural calculus—with at least the following properties:

1. Though not a formal requirement, it helps for the chosen language to be *simple*. By itself reflection is complicated enough that, especially as an initial illustration of the coherence and power of the architecture, it seems

recommended to introduce it into a formalism of minimal internal complexity;
2. It must be possible to access program structures as first-class elements of the language's structural field;
3. Meta-structural primitives must be provided (the ability to *mention* structural field elements, such as data structures and variables, as well as to *use* them); and
4. The underlying architecture should facilitate the embedding, within the calculus, of the procedural components of its own meta-theory.

The second property could be added to a language: we could devise a variant on ALGOL, for example, in which ALGOL programs were made an extended data type, but Lisp already possesses this feature. In addition, since (in the formal semantical analysis presented in following [dissertation] chapters) I will use an extended λ-calculus as the meta-language, it is natural to use a procedural calculus that is functionally oriented. Finally, although full-scale modern Lisps are as complex as any other languages, both Lisp 1.5 and Scheme have the requisite simplicity.

Lisp has other recommendations as well. Because of its support of accessible program structures, it provides considerable evidence of exactly the sort of inchoate reflective behaviour that it has been my aim to reconstruct The explicit use of EVAL and APPLY, for example, provides considerable fodder for subsequent discussion, both in terms of what they do well and how they are confused. In [dissertation] chapter 2, for example, I describe half a dozen types of situation in which a standard Lisp programmer would be tempted to use these meta-structural primitives, only two of which in the deepest sense have anything to do with the explicit manipulation of expressions; the other four, I will argue, ought to be treated directly in the object language—and their use of metastructural

machinery understood to be no more than a "work-around" for fundamental failures in Lisp's original design. And finally, and non-trivially, Lisp is the *lingua franca* of the AI community; this fact alone makes it an eminent candidate. A69

1f-i 1Lisp as a Distillation of Current Practice

The decision to use Lisp as a base does not solve all of cur problems, since the name "Lisp" still refers to a wide range of languages and dialects. For purposes of this dissertation it has seemed simplest to define a simple kernel, not unlike Lisp 1.5, as a basis for further development, in part to have a fixed and well-defined target to set up and criticise, and in part so that I can collect into one dialect the features that prove most important for subsequent analysis. I take Lisp 1.5 as the primary source for the result, which I have called 1Lisp, although some facilities I will ultimately want to examine as (often inchoate) examples of reflective behaviour—such as CATCH and THROW and QUIT—have been added to the repertoire of behaviours manifested in McCarthy's original design. Similarly, I have included macros as a primitive procedure type, as well as intensional and extensional procedures of the standard variety ("call-by-value" and "call-by-name," in standard computer science parlance, although I avoid these terms, since I reject the notion of "value" entirely).

It turns out not to be entirely simple to present 1Lisp. given my theoretical biases, since so much of what I will ultimately reject about it comes so quickly to the surface in explaining it. However I have felt that it is important to present this formalism without modification, because of the role I ask it to play in the structure of the overall argument. In particular, my desideratum for the dialect is not that it be clean or coherent, but rather that it serve as a vehicle in which to examine a body of practice suitable for subsequent reconstruction. To the extent that I make empirical claims about semantic reconstruction, I

use 1Lisp as evidence in its role as being a model of all extant Lisp practice. It is therefore theoretically critical, given this role, that I leave this practice as intact as possible, free of my own theoretical biases. Even though it is a dialect of my own design, therefore, I have intentionally but uncritically forged it in terms of received notions of evaluation, lists, free and global variables, and so forth.

As an example of the style of analysis to be engage in, figure 16 gives a diagram of the 1Lisp category structure—to be contrasted with the category structure of 2Lisp and 3Lisp, which

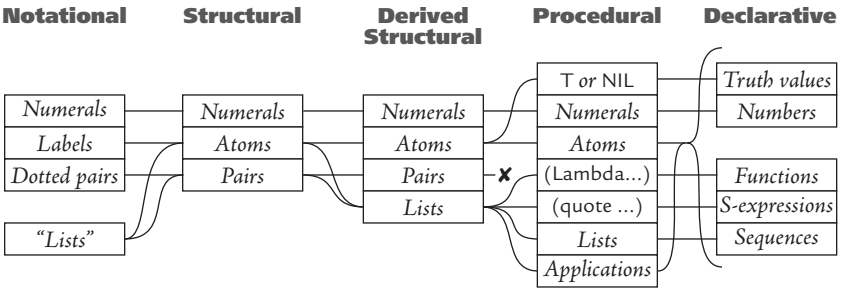

Figure 16 — The Category Structure of Lisp 1.5

has been designed to satisfy the category alignment mandate. The intent of the diagram is to show that in 1Lisp (as in any computational calculus) there are a variety of ways in which structures or s-expressions may be categorised—represented in turn by each of the vertical columns. The point I am attempting to demonstrate is the (unnecessary) complexity of interaction between these various categorical decompositions.

Consider each of these various 1Lisp categories in brief. The first column *(notational)* is categorised by the lexical categories accepted by the reader (including strings that are parsed into notations for numerals, lexical atoms, and "list" and "dotted-pair" notations for pairs). Another categorisation *(structural)* is in terms of the primitive types of s-expression (numerals,

atoms, and pairs); this is the categorisation typically revealed by the primitive structure typing predicates (in ILisp I call this procedure TYPE, but it is traditionally encoded in an amalgam of ATOM and NUMBERP). A third traditional categorisation *(derived structure)* includes not only the primitive s-expression types but also the derived notion of a *list*—a category built up from some pairs (those whose CARS are, recursively, lists) and the atom NIL. A fourth taxonomy (labeled *procedural consequence)* is embodied by the primitive processor: thus ILisp's evaluation processor (EVAL) sorts structures into various categories, each handled differently. This is the "dispatch" categorisation that one typically finds at the top of metacircular definitions of EVAL and APPLY. In most Lisp metacircular processors six categories are discriminated:

1. The self-evaluating atoms T and NIL;
2. The numerals;
3. The other atoms, used as variables or global function designators, depending on context;
4. Lists whose first clement is the atom LAMBDA, used to encode applicable functions;
5. Lists whose first clement is the atom QUOTE; and
6. Other lists, which in evaluable positions represent function application.

Finally, the fifth taxonomy *(declarative import)* has to do with declarative semantics—i.e., discriminates categories of structure based on their *signifying* different sorts of semantic entities. Once again a different category structure emerges: applications and variables can signify semantic entities of arbitrary type *except that they cannot designate procedures* (since ILisp is first-order); the atoms T and NIL signify Truth and Falsity; general lists, plus again (in different contexts) the atom NIL, signify enumerations (sequences): the numerals signify numbers; and so on and so forth.

The reason why the demerits of this non-alignment of categories multiply in a reflective dialect is that reflective programs need to know about all of them, in different situations and for different purposes—and also about the relationships between and among them (as, impressively, all human Lisp programmers do). And remember, too, that as one climbs from reflective level 1 to yet higher reflective levels, the combinatorics of non-alignment would multiply correspondingly. I need not dwell on the evident disarray that would likely result.

One other example of 1Lisp behaviour will be illustrative. I have mentioned above that 1Lisp requires the explicit use of APPLY in a variety of circumstances. These include the following:

1. When an argument expression designates a function *name*, rather than a function—as for example in

   (APPLY (CAR '(+ - *)) '(2 3))

2. When the arguments to a multiple-argument procedure are designated by a single term, rather than designated individually. Thus if X evaluates to the list (3 4), one must use (APPLY '+ X) rather than (+ X) or (+ . X).
3. When a function is designated by a variable rather than by a global constant. Thus one must use:

   (LET ((FUN '+)) (APPLY FUN '(1 2)))

   rather than the simpler:

   (LET ((FUN '+)) (FUN 1 2))

4. When the arguments to a function are "already evaluated," since APPLY, although itself extensional (it is an "EXPR"), does not re-evaluate the arguments even if the procedure being applied is an EXPR. Thus one uses:

```
(APPLY '+ (LIST X Y))
```

rather than:

```
(EVAL (CONS '+ (LIST X Y)))
```

As I will show, in 2Lisp and 3Lisp only the first of these will require explicitly mentioning the processor function by name, because it inherently deals with the *designation of expressions*, rather than with the designation of their referents. Because of their category alignment, 2Lisp and 3Lisp treat the other three cases adequately in the object language.

1f-ii The Design of 2Lisp

Though it meets the criterion of simplicity, 1Lisp provides more than ample material for further development, as the previous examples suggest. Once I have introduced it, as mentioned earlier, I subject it to a semantical analysis that leads us into an examination of computational, semantics in general, as described in the previous section. The search for semantical rationalisation, and the exposition of the 2Lisp that results, occupies a substantial part of the dissertation, even though the resulting calculus still fail to meet the requirements of procedural reflection (as befitting the underlying thesis that reflection is relatively straightforward, once these semantical issues are taken care of). I discussed what semantic rationalisation comes to in general in a previous section (§1.f.1); here I sketch how its mandates are embodied in the design of 2Lisp.

The most striking difference between 1Lisp and 2Lisp is that the latter rejects evaluation in favour of independent notions of *simplification* and *reference*. Thus, 2Lisp's processor is not called EVAL, but NORMALISE, where by *normalisation I* refer to a particular form of expression simplification that takes each structure into what I call a *normal-form* designator of that expression's referent (making normalisation designation-pre-

serving). Details are provided in [dissertation] chapter 4, but a sense of the resulting architecture can be given here. A70

Simple object level computations in 2Lisp (those that do not involve meta-structural structures designating other elements of the Lisp field) are treated in a manner that looks very similar to 1Lisp. The expression (+ 2 3), for example, normalises to 6, and the expression (= 2 3) to $F (the primitive 2Lisp boolean constant designating falsity). On the other hand an obvious superficial difference is that in 2Lisp *meta-structural* terms are not automatically dereferenced. Thus the quoted term 'X, which in 1Lisp would evaluate to X, normalises in 2Lisp to itself (that is: to 'X). Similarly, whereas (CAR '(A . B)) would evaluate in 1Lisp to A, in 2Lisp it normalises to 'A. Similarly, in 1Lisp (CONS 'A 'B) evaluates to the pair (A . B); in 2Lisp the corresponding expression would normalise to the handle '(A . B).

From these almost trivial examples, one might be tempted to embrace the following idea: that the 2Lisp processor is just like the 1Lisp processor, except that it puts a quote back on before returning the result. But that reading is ill-advised; the difference, more theoretically motivated, is more substantial in terms of structure, procedural protocols, and semantics. For starters 2Lisp, like Scheme, is statically-scoped and higher-order; function-designating expressions may be passed as regular arguments. 2Lisp is also structurally different from 1Lisp; there is no derived notion of *list*, but rather a primitive data structure called a **rail** that serves the function of designating a sequence of entities (pairs are still used to encode function applications). What in 1Lisp are called "quoted expressions" correspond to the primitive structural type **handle**, not to applications framed in terms of a (pseudo) QUOTE procedure; they are also canonical (one per structure designated). The 2Lisp notation 'X, in particular, is not an abbreviation for (QUOTE X,), but rather the primitive notation for the handle that is the unique normal-form designator of the atom X. There are other

notational differences as well: rails are expressed with square brackets (thus the expression '[1 2 3]' notates a rail of three numerals that in turn designates a sequence of three numbers), and expressions of the form

$$(F\ A_1\ A_2\ \ldots\ A_k)$$

expand not into

$$(F\ .\ (A_1\ .\ (A_2\ .\ (\ldots\ .\ (A_k\ .\ \text{NIL})\ldots))))$$

but instead into

$$(F\ .\ [A_1\ A_2\ \ldots\ A_k])$$

The category structure of 2Lisp is summarised in figure 17.

Closures, which have historically been treated as rather curious entities somewhere in between functions and expressions, emerge in 2Lisp as standard expressions; in fact I *define* the term '**closure**' to refer to a *normal-form function designator*. Not only are closures pairs, but all normal-form pairs are A71 closures, illustrating once again the category alignment that permeates the design.

As stated above, all 2Lisp normal-form designators are not only *stable* (self-normalising), but also *side-effect free* and *context-independent*. A variety of facts emerge from this result. First, the primitive processor procedure (NORMALISE) can be proved to be *idempotent* in terms of both result and total effect:

$$\forall S\ [\ (\text{NORMALISE}\ S) = (\text{NORMALISE}\ (\text{NORMALISE}\ S))\ ] \quad [10]$$

Consequently, as in the λ-calculus, the result of normalising a constituent (in an extensional context) in a composite expression may be substituted back into the original expression, in place of the non-normalised expression, yielding a *partially simplified* expression having the same designation and same normal-form as the original. So support for "partial evalua-

tion" is in some sense an automatic feature of the two dialects. In addition, in code-generating code such as macros and debuggers and so forth, there is no need to worry about whether an expression has *already* been processed, since second and subsequent processings will never cause any harm (nor, as it happens, will they take any time).

All of the foregoing facts can in some sense be considered to be *simplifications* embedded in the design of 2Lisp. Most of 2Lisp's complexities emerge only when one consider forms

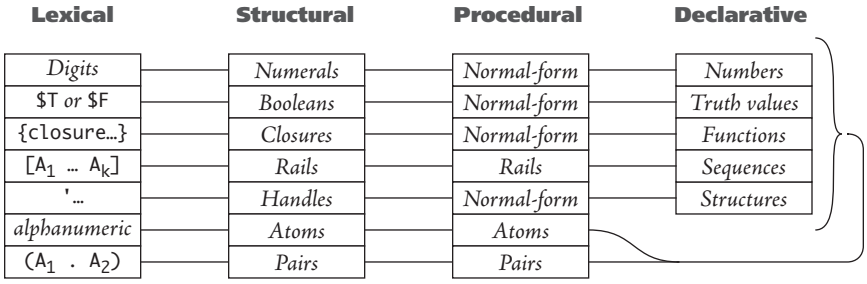

Figure 17 — The Category Structure of 2Lisp (and 3Lisp)

that designate other semantically significant forms. The intricacies of such "level-crossing" expressions are the stock-in-trade of a reflective system designer, and only by setting such issues straight *before* we consider reflection proper will we face the latter task adequately prepared.

Primitive procedures called NAME and REFERENT (notationally abbreviated '↑' and '↓', respectively) are provided to mediate between sign and significant (they must be primitive because without them the processor provably remains semantically flat); thus (taking '⇒' to mean "normalises to"):

↑3 ≡ (NAME 3) ⇒ '3
↓''A ≡ (REFERENT ''A) ⇒ 'A

The issue of the explicit use of APPLY, mentioned in the discussion of 1Lisp, above, is instructive to examine in the 2Lisp context, since it manifests both the structural and the semantic differences between 2Lisp and its precursor dialect. In 1Lisp, the functions EVAL and APPLY mesh in a well-known mutually-recursive fashion. Evaluation is uncritically thought to be defined over *expressions,* but it is much less clear what application is defined over. On one view, APPLY is a functional that maps functions and (sequences of) arguments onto the value of the function at that argument position—thus making it a second (or higher) order function. On another view, APPLY takes two *expressions* as arguments, and has as its value a third expression that *designates* the value of the function designated by the first argument at the argument position designated by the second. In 2Lisp I will call the first of these notions **application** and the second **reduction** (the latter in part because the word suggests an operation over expressions, and in part by analogy with the β-reduction of Church.[15] Current Lisp systems are less than lucid regarding this distinction (in MacLisp, for example, the *function* argument is an expression, whereas the *arguments* argument is not an expression, nor is the value). The position I will adopt is depicted in figure 18 (to be explained more fully in [dissertation] chapter 3).

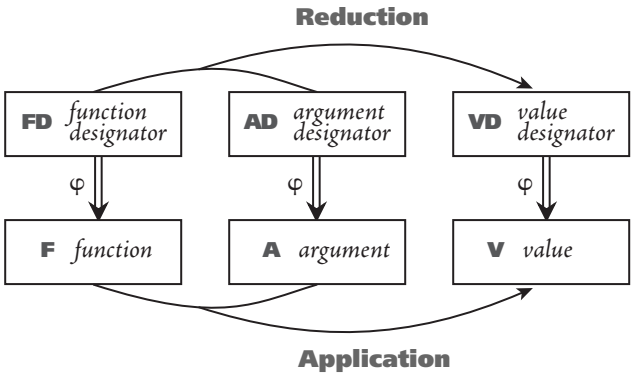

Figure 18

15. Church (1941).

The procedure REDUCE, together with NORMALISE will of course play a major role in the characterisation of 2Lisp, and in the subsequently constructed reflective 3Lisp. It is worth noting, however, that although it would be trivial to do so, there is no reason to define a designator of the APPLY function, since any term of the form:

(APPLY FUN ARGS)

would be equivalent in both designation and effect (i.e., would be equivalent in full computational significance) to:

(FUN . ARGS)

In contrast, since it is a meta-structural function, REDUCE is neither trivial to define (as is APPLY) nor recursively empty.

By way of summary, we can list the following as the most salient distinctions between 2Lisp and 1Lisp:

1. **Scoping:** 2Lisp is lexically scoped, in the sense that variables free in the body of a LAMBDA form take on the bindings in force in their statically enclosing context, rather than from the dynamically enclosing context at the time of function application.
2. **Functions:** Functions are first-class semantical objects, and may be designated by standard variables and arguments. As a consequence, the function position in an application (the CAR of a pair) is both procedurally and declaratively "extensional," and thus normalised in exactly the same way as argument positions.
3. **Processing:** Evaluation is rejected in favour of independent notions of *simplification* and *reference*. The primitive processor is a particular kind of *simplifier*. rather than being an *evaluator*. In particular, it *normalises* expressions, returning for each input expression a normal-form co-designator.

4. **Declarative Semantics:** A complete theory of declarative semantics is postulated for all s-expressions. prior to and independent of the specification of how they are treated by the processor function—a pre-requisite to the claim that the processor is designation-preserving).
5. **Closures:** Closures—normal-form function designators—are valid and inspectable s-expressions.
6. **Normal Form:** Though not all normal-form expressions are canonical (functions, in particular, may have arbitrarily many distinct normal-form designators), nevertheless they are all stable (self-normalising), side-effect free, and both declaratively and procedurally context independent.
7. **Semantically Flat:** The primitive processor (designated by NORMALISE) is semantically flat; in order to shift level of designation one of the explicit semantical primitives NAME (↑) or REFERENT (↓) must be applied.
8. **Category Alignment:** 2Lisp is category-aligned (as indicated in figure 17, above): there are two distinct structural types, pairs and rails, that respectively encode function applications and sequence enumerations. There is in addition a special two-element structural class of boolean constants. There is no distinguished atom NIL.
9. **Binding:** Variable binding is *co-designative*, rather than being either *evaluative* or *designative*, in the sense that a variable normalises to what it is bound to, and therefore designates the referent of the expression to which it is bound. Although I will speak of the binding of a variable, and of the referent of a variable, I will not speak of a variable's *value*, since that term conflates these two notions.
10. **Identity:** Identity considerations on normal-form des-

ignators are as follows: the normal-form designators of truth-values, numbers, and s-expressions (the booleans, numerals, and handles, respectively) are unique. Normal-form designators of sequences (rails) and functions (pairs) are not. No atoms are normal-form designators of anything; therefore the question does not arise in their case.

11. **LAMBDA:** The use of LAMBDA is purely an issue of abstraction and naming, and is completely divorced from procedural *type* (extensional, intensional, macro, and so forth). A72

* * *

As soon as I have settled on the definition of 2Lisp, however, I will begin to criticise it. In particular, I will provide an analysis of how 2Lisp fails to be appropriately reflective, in spite of its semantical cleanliness.

A number of problems with 2Lisp in particular emerge as troublesome. First, it will turn out that the clean *semantical* separation between meta-levels is not yet matched with a clean *procedural* separation. For example, too strong a separation between environments, with the result that intensional procedures become extremely difficult tn use, shows that in one respect, 2Lisp's inchoate reflective facilities suffer from insufficient causal connection. On the other hand, awkward interactions between the control stacks of inter-level programs will show how, in other respects, there is *too much* connection. In addition, although I will demonstrate a metacircular implementation of 2Lisp in 2Lisp, and will provide 2Lisp with explicit names for its basic interpreter functions (NORMALISE and REDUCE), these two facilities will remain utterly unconnected—an instance of a general problem to be discussed in [dissertation] chapter 3 on reflection in general.

1f-iii The Procedurally Reflective 3Lisp

From this last analysis will emerge the design of 3Lisp, a procedurally reflective Lisp and the last of the dialects to be considered here.

As presented in [dissertation] chapter 5, 3Lisp differs from 2Lisp in a variety of ways.

1. The fundamental reflective act is identified and accorded tbe centrality it deserves in the underlying language definition.
2. Each reflective level is granted its own environment and continuation structure, with the environments and continuations of the levels below it accessible as first-class objects (inheriting a Quinean stamp of ontological approval, since they can be the values of bound variables). A73
3. As mentioned in the earlier discussion these environments and continuations are theory relative. The (procedural) theory is embodied in the 3Lisp reflective model, a causally connected variant on the metacircular interpreter of 2Lisp discussed in §1.e.
4. Surprisingly, the integration of reflective power into the metacircular—now reflective—model is itself extremely simple (though to *implement* the resulting machine is not trivial).
5. Reflecting its more complete nature, in a number of ways 3Lisp is notably *simpler* than 2Lisp.

Once all these moves have been taken it will be possible to merge the explicit reflective version of NORMALISE and REDUCE, and the similarly named primitive functions. In other words the 3Lisp reflective model unifies what in 2Lisp were separate: primitive names for the underlying processor, and explicit metacircular programs demonstrating the procedural structure of that processor.

It was a consequence of defining 2Lisp in terms of NORMALISE, a species of simplification, that the 2Lisp processor is "semantically flat": the semantical level of an input expression is always the same as that of the expression to which it simplifies.. An even stronger claim holds for function application. Except in the case of the explicit level-shifting functions NAME (↑) and REFERENT (↓), the semantical level of the result is also the same as that of all of the arguments. This is all evidence of the effort to drive a wedge between simplification and *dereferencing* mentioned earlier. 3Lisp inherits this semantical characterisation; note that it remains true *even in the case of reflective functions*. A74

A semantically-flat (fixed-level) processor of this form—one of the reasons 2Lisp was designed this way—enables an important move: it becomes possible, though only in an approximate sense, to identify *declarative meta levels* with *procedural reflective levels*. This does not quite have the status of a *claim*, because it is virtually mandated by the Knowledge Representation Hypothesis (furthermore, the correspondence is somewhat asymmetric: declarative levels can be crossed within a given reflective level, but reflective shifts always involve shifts of designation). But it is instructive to realise that we have been able to identify the reflective act (that makes available the structures encoding the processing state and so forth) with two shifts: (i) the shift from objects to their names, and (ii) the shift from tacit aspects of the background to objects. *Reification*, that is, *emerges as the first form of actively engaged semantic ascent*. Thus: (i) what was *used* prior to reflection is *mentioned* upon reflecting; (ii) what was *tacit* prior to reflection becomes *used* upon reflection. When this behaviour is A75 combined with the ability for reflection to recurse, we are able to lift structures that are normally tacit into explicit view in one simple reflective step; we can then obtain access to designators of those structures in another.

Later in the dissertation both the 3Lisp reflective model, and a MacLisp implementation of it, will be provided by way of definition. In addition, some hints will be presented of the style of semantical equation that would be required for a traditional denotational-semantics style account of 3Lisp—though it is important to admit that a full semantical treatment of procedural reflection in general or of 3Lisp in particular has yet to be worked out.

In a more pragmatic vein, however, and in part to show how 3Lisp satisfies many of the *desiderata* that motivated the original definition of the concept of reflection, I will present a number of examples of programs defined in 3Lisp: a variety of standard functions that make use of calls to the processor, access to the implementation (debuggers, "single-steppers," and so forth), and non-standard "evaluation" (processing) protocols. The suggestion will be made that the case with which these powers can be embedded in "pure" programs recommends 3Lisp as a plausible dialect in its own right. Nor is this simply a matter of using 3Lisp as a theoretical vehicle in which to model or implement these various constructs, or of showing that such models fit naturally and simply into the 3Lisp dialect (as a simple continuation-passing scheme can for example be shown to be adopted in Scheme). The claim is stronger: that such functionality can be naturally embedded in 3Lisp in a manner that allows it to be *congenially mixed* (without pre-processing or pre-compilation) *with simpler, more standard forms of practice*. Without the user normally having to use (or even understand) explicit continuation-passing style, nonetheless, at any point in the course of the computation, the applicable continuation is easily and explicitly available (upon reflection) for any programs that wish to deal with such things directly. Similar remarks hold for other aspects of the control structure and environment

One final comment about the 3Lisp architecture will relate

it to the two views on reflection—"level-shifting" and "infinite-tower"—mentioned at the end of §1.e. Modulo the amount of time it takes, processing mediated by the 3Lisp reflective model is guaranteed to yield indistinguishable behaviour (at least from a non-reflective point of view—there are subtleties here) from basic, non-reflected processing. It is this fact that allows us to make the abstract claim that 3Lisp runs in virtue of an infinite number of levels of reflective models all running at once. by an (infinitely fleet) overseeing processor running at level ∞. The resulting infinite abstract machine is well defined, for it is of course behaviourally indistinguishable from the perfectly finite 3Lisp that will already have been laid out (and implemented). For some purposes 3Lisp is most easily described in terms of this infinite tower—and in some ways, too, it is the easiest model for the 3Lisp programmer to have in mind, when writing programs. Such a programmer can write programs to be interpreted at any reflective level, and cannot tell that the full infinitude of levels are not being run (the implementation surreptitiously constructs them and places them in view each time the user's program steps back to view them), such a characterisation is usually more illuminating than talk of the processor "switching back and forth from one level to another". In terms of mathematical analysis, treating 3Lisp as a purely formal object, the infinite tower characterisation would also be more likely to be preferred. On the other hand, when taken as a model of psychologically intuitive reflection—based on a vague desire to locate the *self* of the machine at some level or other—the language of level-shifting seems to be more highly recommended. Level-shifting is also a major and constant concern for anyone person who designs and constructs a 3Lisp implementation.

1f·iv Reconstruction Rather Than Design

2Lisp and 3Lisp can claim to be dialects of Lisp only on a gen-

erous interpretation. Both dialects are unarguably more different from the original Lisp 1.5 than are all other dialects that have previously been proposed, including for example Scheme, MDL, NIL, SEUS, MacLISP, InterLISP, and Common Lisp.[16]

In spite of this difference, however, I view it as important to the exercise to call these languages Lisps. The aim in developing them has not been simply to propose some new variants in a grand tradition, perhaps better suited for a certain class of problem than others that have gone before. Rather—and this is one of the reasons that this dissertation is as long as it is—it is my claim that, in spite of their differences from that of standard Lisps,

> *The architecture of these new dialects is a more accurate reconstruction than has heretofore been provided of the underlying coherence that already organises our communal understanding of what Lisp is.*

I am making an empirical claim, in other words—a claim that should ultimately be judged as right or wrong. Whether 2Lisp or 3Lisp are *better* than previous Lisps is of course a matter of interest on its own, but it is not the thesis that this dissertation has set out to argue.

1g Remarks

1g·i Comparison with Other Work

Although I know of no previous attempts to construct eitller a semantically rationalised or a reflective computational calculus, the research presented here is of course dependent on, and related to, a large body of prior work. There are in particular

16. Scheme is reported in Sussman and Steele (1975) and in Steele and Sussman (1978a); MDL in Galley and Pfister (1975), NIL in White (1979), MacLISP in Moon (1974) and Weinreb & Moon (1981), and InterLISP in Teitelman (1978). Common Lisp and SEUS are both under development, as this is being written, and have not yet been reported in print, so far as I know (personal communication with Guy Steele and Richard Weyhrauch).

four general areas of study with which this project is best compared:

1. Investigations into the meta-cognitive and intensional aspects of problem solving (this includes much current research in Artificial Intelligence);
2. The design of logical and procedural languages (including virtually all of programming language research, as well as the study of logics and other declarative calculi);
3. General studies of semantics (including both natural language and logical theories of semantics, and semantical studies of programming languages); and
4. Studies of self-reference, of the sort that have characterised much of metamathematics and the theory of computability throughout this century, particularly since Russell, and including the formal study of the paradoxes, the Gödel incompleteness results, and so forth.

I will make detailed comments about connections between this project and such other work throughout the discussion (for example in [dissertation] chapter 5 I will compare the reflective sense of "self-reference" with the notion traditionally studied in logic and mathematics), but some general comments can be made here.

Consider first the meta-cognitive aspects of problem-solving, of which the dependency-directed deduction protocols presented by Stallman and Sussman, Doyle, McAllester, and others are an illustrative example.[17] This work depends on explicit encodings, in some form of meta-language, of information about object-level structures, used to guide a deduction process. Similarly, the meta-level rules of Davis in his TEIRESIUS system,[18] and the use of meta-levels rules as an aid in planning,[19] can be viewed as examples of inchoate reflective

17. Stallman and Sussman (1977), de Kleer et al. (1977).
18. Davis (1980)
19. Stefik (1981a and 1981b).

problem solvers. Some of these expressions are primarily procedural in intent,[20] although declarative statements (for example about dependencies) are perhaps more common, with respect to which particular procedural protocols are defined.

The relationship of the current project to this type of work is more one of support than of direct contribution. I do not present (or even hint at) problem solving strategies involving reflective manipulation, although the fact that others are working in this area has certainly been a motivation for my research. Rather, I attempt to provide a rigorous account of the particular issues that have to do simply with providing *facilities for reflection*, independent of *what such facilities are then used for*. An analogy might be drawn to the development of the λ-calculus, recursive equations, and Lisp, in relationship to the use of these formalisms in mathematics, symbolic computation, and so forth: the former projects provide a language and architecture, to be used reliably, and perhaps without much conscious thought, as the basis for a wide variety of applications. The present dissertation will be successful not if it forces everyone working in meta-cognitive areas to think about the architecture of reflective formalisms, but almost the opposite: if it allows them to *forget* that the technical details of reflection were ever considered to be problematic. Church's α-reduction was a successful manoeuvre precisely because it means that one can treat the λ-calculus in the natural way; I hope that my treatment of reflective procedures will enable those who use 3Lisp or any subsequent reflective dialect to treat "backing-off" in what they take to be "the natural way."

The "reflective problem-solver" reported by Doyle[21] deserves a special comment. Again, I provide an underlying architecture which might facilitate his project, without actually contributing solutions to any of his particular problems about how reflection should be effectively used, or when its deploy-

20. de Kleer et al. (1977).
21. Doyle (1981).

ment is appropriate. Doyle's envisaged machine is a full-scale problem solver; it is also (so at least he argues) presumed to be large, to embody complex theories of the world, and so forth. In contrast, 3Lisp is not a problem solver at all (all the user is "given" is a language—very much in need of programming); it embodies only a small procedural theory of itself, and it is really quite small. As well as these differences in goals there are differences in content (I for example endorse a set of reflective levels, rather than any kind of true instantaneous self-referential "reflexive" reasoning); it is difficult, however, to determine with very much detail what his proposal comes to, since his report is more suggestive than final.

Given that 3Lisp is not a problem solver of the sort Doyle proposes, it is natural to ask whether it would be a suitable language in which Doyle might *implement* his system. There are two different kinds of answer to this question, depending on how he takes his project.

If, on the one hand, Doyle is proposing a design of a complete computational architecture (i.e., a process reduced in terms of an ingredient processor and a structural field), and wishes to *implement* it in some convenient underlying language, then 3Lisp's reflective powers will not in themselves immediately engender corresponding reflective powers in the virtual machine that he implements. Reflection, as I have been at considerable pains to demonstrate, is first and foremost a semantical phenomenon, and *semantical properties*—designation and normalisation protocols and reflection and the rest—*do not cross implementation boundaries* (this is one of the great powers, but also a very serious limitation, of implementation). A76 3Lisp would be useful in such a project to the extent that it is generally a useful and powerful language, but it is important to recognise that its reflective powers cannot be used directly to provide reflective capabilities in other architectures implemented on top of it.

There is an alternative strategy open to Doyle, however, by which he could use 3Lisp's reflective powers more directly. If, rather than defending a generic reflective architecture, he more simply intended to show how a particular kind of reflective reasoning was useful, he could perhaps construct such behaviour in 3Lisp, and thus use its reflective capabilities rather directly. There are consequences of this approach, however: he would have to accept 3Lisp structures *and semantics*, including among other things the fact that it is purely a procedural formalism. It would not be possible, in other words, to encode a full descriptive language on top of 3Lisp, and then use 3Lisp's reflective powers to reflect in the general sense with these descriptive structures. If one aims to construct a general or purely descriptive formalism, one would have to make that architecture reflective on its own.

None of these conclusions stand as criticisms of 3Lisp. They are entailed by fundamental facts about computation and semantics—not limitations of the particular theory or dialect I propose (i.e., they would, and necessarily so, be equally true of any other proposed architecture).

This is one reason, among many, why I view 3Lisp not as *the contribution made in this dissertation*, but rather as *an example to exhibit its contribution: the conceptual structure of how to design and build a reflective architecture*. Thus it is my hope that what would be useful from this dissertation for Doyle, or for anyone else in a parallel circumstance, is the detailed structure of a reflective system that I have attempted to explicate here—an architecture and a concomitant set of theoretical terms *to help such a person analyse and structure whatever architecture they design, adopt, or embrace*. Thus I would count the present contribution a success if it proved useful, for Doyle or anyone else, to make use of:

1. The φ/ψ distinction;

2. The relationship between semantical levels and reflective levels;
3. The encoding of the reflective model within the calculus;
4. The strategy of adopting a virtually infinite tower of processors as an ideal in terms of which to define a finite model of level-shifting;
5. The semantic flatness and uniformity of a normalising processor;
6. The elegance of category-alignment;

And so forth. It is in this sense that I hope that the theory and understanding that 3Lisp embodies will contribute to A77 problem-solving research (and to programming language research), rather than the particular formalism I have developed and demonstrated by way of illustration.

The second type of research with which this project has strong ties is the general tradition of providing formalisms to be used as languages and vehicles for a variety of other projects—including the formal statement of theories, the construction of computational processes, the analysis of human language, and so forth. I take this tradition to be sufficiently broad (in particular, to include logic and the λ-calculus, plus virtually all programming language research) that it is difficult to say very much that is specific, though a few comments can be made.

First, I of course owe a tremendous debt to the Lisp tradition in general,[22] and also to the recent work of Steele and Sussman.[23] Particularly important is their Scheme dialect—in many ways the most direct precursor of 2Lisp (In an early version of the dissertation I called Scheme "1.7-Lisp," since it

22. References to specific Lisp dialects are given in note 16, above; more general accounts may be found in Allen (1978), Weisman (1967), Winston and Horn (1981), Charniak et al. (1980), McCarthy et al. (1965), and McCarthy and Talbott (forthcoming).
23. Steele (1976), Steele & Sussman (1976, 1978b).

takes what I see as approximately half of the step from Lisp 1.5 to the semantically rationalised 2-Lisp). Second, my explicit attempt to unify the declarative and procedural aspects of this tradition has already been mentioned—a project that is (as far as I know) without precedent. Note, as mentioned in the Introduction, that I do not consider Prolog[24] to count as having done this, since it provides two calculi together, rather than presenting a single calculus under a unified theory. Finally, as documented throughout the text, inchoate reflective behaviour can be found in virtually all comers of computational practice; the Smalltalk language,[25] to mention just one example, includes a meta-level debugging system which allows for the inspection and incremental modification of code in the midst of a computation.

The third and fourth classes of previous work listed above have to do with general semantics and with self-reference. The first of these is considered explicitly in [dissertation] chapter 3, where I compare my approach to this subject with model theories in logic, semantics of the λ-calculus, and the tradition of programming language semantics; no additional comment is required here. Similarly, the relationship between the notion of reflection I present and traditional concepts of self-reference are taken up in more detail in [dissertation] A78 chapter 5; here I merely comment that my concerns, perhaps surprisingly, are constrained almost entirely to computational formalisms. Unless a formal system embodies a *locus of active agency*—an internal processor (i.e., process) of some sort—the entire question of causal relationship between an encoding of self-referential theory and what I consider a genuine reflective model cannot even be asked.

We often informally think of a natural deduction "process" or some other kind of deductive apparatus making inferences

24. Clark and McCabe (1979), Roussel (1975), and Warren et al. (1977).
25. Goldberg (1981); Ingalls (1978).

over first-order sentences, as a heuristic in terms of which to make sense of the formal notion of derivability. Strictly speaking, however, in the purely declarative tradition *derivability* is no more than a *formal relationship that holds between certain sentence types*; no activity is involved. There are no notions of *next* or of *when* a certain deduction is made. *If* one were to specify an active deductive process over such first-order sentences, then it is imaginable that one could include sentences (relative to some axiomatisation of that deductive process) in such a way that the operations of the deductive process were appropriately controlled by those sentences (this is the suggestion explored briefly in §1·b·ii). The resulting machine, however—not merely in its reflective incarnation, but even prior to that, by including an active agency—cannot fairly be considered simply *logic*, but rather a full computational formalism of some sort.

Needless to say, I believe that a reflective version of such a descriptive system could be built (in fact it is my intent to develop just such an architecture in the future). My position A79
with respect to such an image rests on two observations: (i) the result would be an inherently *computational* artefact, in virtue of the addition of independent agency, and (ii) 3Lisp, although reflective, is not yet such a formalism, since it is purely procedural.

I conclude with one final comparison. The formalism closest in spirit to 3Lisp is Richard Weyhrauch's FOL system,[26] although my project differs from his in several important technical ways. First, like Doyle's system, FOL is a problem solver: it embodies a theorem-prover, although it is possible (through the use of FOL's meta-levels) to give it guidance about the deduction process. In spite of those facilities, however, FOL is not a *programming language*. Furthermore, FOL adopts—in fact explicitly endorses—the distinction between declarative and procedural

26. Weyhrauch (1978).

languages (first order logic and Lisp, in particular), using the procedural calculus as a *simulation structure* rather than as a descriptive or designational language. Weyhrauch claims that the power that emerges from combining—but maintaining as distinct—these "language-simulation-structure" pairs, as he calls them ("L-S pairs"), at each level in his meta hierarchy, is one of his primary contributions. It is my own claim, in contrast, that the greatest power will arise from *dismantling* the difference between procedural and declarative calculi.

There are other differences as well. I take the interpretation function that maps terms onto objects in the world outside the computational systems (φ) to be foundational. It would appear in Weyhrauch's systems as if that particular semantical relationship is abandoned in favour of internal relationships between one formal system and another. A more crucial distinction is hard to imagine—though there is some evidence[27] that this apparent difference may have to do with our respective uses of terminology, rather than with deep ontological or epistemological beliefs.

In sum, FOL and 3Lisp are technically quite distinct, and the theoretical analyses on which they are based almost unrelated. At a more abstract level, however, they are clearly based on similar—and perhaps parallel, if not identical—intuitions. Furthermore, I would argue that 3Lisp represents merely a first step in the development of a fully reflective calculus based on a fully integrated theory of computation and representation; how such an eventual system, once it were defined, would differ from FOL remains to be seen. It seems likely that the resulting unified calculus, rather than the dual-calculus nature, would be the most obvious technical distinction, although the actual structure of the descriptive language, semantical metatheories, and so forth, are also likely to differ both in substance and in detail.

27. I am indebted to Richard Weyhrauch for personal communication on these points.

One remaining difference is worth exploring in part because it reveals a deep but possibly distinctive character of my treatment of Lisp. It is clear from Weyhrauch's system that he considers the procedural formalism to represent a kind of *model* of the world—in the sense of an (abstract) artefact whose structure or behaviour mimics that of some other world of interest. Under this approach the computational behaviour can be taken *in lieu of* or *in place of* the real behaviour in the world being studied. Consider for example the numeral addition that is the best approximation a computer can make to actually "adding numbers" (whatever that might be). When we type '(+ 1 2)' into a Lisp processor, and it returns '3', we are liable to take those numerals not so much as *designators* of the respective numbers, but instead as *models*. There is no doubt that the input expression '(+ 1 2)' is a linguistic artefact; on the view I will adopt in this dissertation there is no doubt that the resultant numeral '3' is also a linguistic artefact. I do want to admit, however, that there is a not unnatural tendency to think of the latter as "standing in place of" the actual number, in a different sense from standard designation or naming. It is this sense of *simulation* rather than *description* that, as far as I A80 understand it, underlies Weyhrauch's use of Lisp.

I fundamentally believe that this is a limited view, however—and go to considerable trouble to maintain an approach in which all computational structures are taken to be semantical in something like a linguistic sense, rather than (being taken as) serving as models. Many issues are involved—having to do with such issues as truth, completeness, and so forth—that a simulation stance cannot deal with. At worst, moreover, adopting a simulation stance can lead to a view of computational models that runs in danger of being either radically solipsistic or even, I believe, nihilist. It is exactly the *connection* between a computational system and the world that motivates my entire approach; a connection that I believe can be ignored

only at considerable peril. I in no way rule out computations that in different respects mimic the behaviour of the world they are about; it is clear that certain forms of human analysis involve just this kind of thinking ("stepping through" the transitions of some mechanism in one's head, for example, to "be sure that one understands it"). My point is only that such simulation is *still a kind of thinking about the world*; it is not the A81
world being thought about.

1g·ii The Mathematical Meta-Language

Throughout the dissertation I will employ an informal meta-language, built up from a rather eclectic combination of devices from quantificational logic, the λ-calculus, and lattice theory, extended with some straightforward conventions (such as expressions of the form "if P then A else B" as an abbreviation for "[P ⊃ A] ⋀ [¬P ⊃ B]"). Notationally I will use set-theoretic devices (union, membership, etc.), but these should be understood as defined over *domains* in the Scott-theoretic sense, rather than over unstructured sets. The notations should by and large be self-explanatory; a few standard conventions worth noting are these:

1. '[A → B]' refers to the domain of continuous functions from A to B;
2. '$F : [A \to B]$' means that F is a function whose domain is A and whose range is B;
3. '$<s_1, s_2, \ldots s_k>$' designates the mathematical sequence consisting of the designata of 's_1','s_2', … 's_k';
4. 's^i' refers to the i'th element of s, assuming that s is a sequence (thus $<A, B, C>^2$ is B);
5. '[s × R]' designates the (potentially infinite) set of all tuples whose first member is an element of s and whose second member is an element of R;

6. 'A^*' refers to the power domain of A:

$$[\, A \cup [A \times A] \cup [A \times A \times A] \cup \ldots \,]$$

7. Parentheses and brackets are used interchangeably to indicate scope and function application in the standard way.
8. Standard currying is employed to deal with functions of several arguments. Thus:

$\lambda A_1, A_2, \ldots A_k \,.\, E$ means $\lambda A_1.[\lambda A_2.[\ldots.[\lambda A_k \,.\, E]\ldots]]$
$\lambda \langle A_1, A_2, \ldots A_k \rangle \,.\, E$ means $\lambda A_1.[\lambda A_2.[\ldots.[\lambda A_k \,.\, E]\ldots]]$
$F(B_1, B_2, \ldots B_k)$ means $((\ldots((F(B_1))B_2)\ldots)B_k)$

If I wanted to be more precise, I would be stricter about the use of domains rather than sets, in order that function continuity be maintained, and so forth. It is not my intent here to make the mathematics rigorous, but I trust that it would be straightforward, given the accounts I set down, to take this extra step towards formal adequacy.

1g·iii Examples and Implementations

A considerable number of examples are presented throughout the dissertation, which can be approximately divided into two groups: (i) formal statements about Lisp and about semantics, expressed in the meta-language; and (ii) illustrative programs and structures expressed in Lisp itself (most of the latter are in one of the three Lisp dialects I define, though a few are in standard dialects as well). As the preceding discussion suggests, the meta-linguistic characterisations have not been checked by formal means for consistency or accuracy; the proofs and derivations were generated by the author using paper and pencil. The program examples, on the other hand, were all tested on A82 computer implementations of 1Lisp, 2Lisp, and 3Lisp developed in the MacLisp and "Lisp Machine" Lisp dialects of Lisp

at MIT (a complete program listing of the third of these—a MacLISP implementation of 3-Lisp—is given in the Appendix to this dissertation). Thus, although the examples in the text were typed in by the author as text—i.e., the lines of characters in this document are not actual photocopies of computer interaction—each was nevertheless verified by these implementations. However the implementation presented in the Appendix is a photocopy of the actual computer program listing. Any residual errors (it is hard to imagine every one has been eliminated) must have arisen either from typing errors or from mistakes in the implementation itself. A83

Annotations[1]

A1 ·31/-1/5 It would also have been correct, and philosophically more expected, had this been written "adequate theories of *intentionality*"—i.e., theories of intentionality-with-a-t,[2] the full gamut of issues involved in how it is that a sentence or structure or event α can be meaningful or about something. I did however intend the more specific intensionality-with-an-s.

As the discussion throughout this chapter makes evident, and as is highlighted in dissertation §4c.i,[3] in any reflective system of the sort envisaged one needs to deal not only with *extensional* issues, having to do with the reference, denotation, or designation of symbols and other intentional entities, but also with usually finer-grained *intensional* notions of *what they mean*. As usual, the issue comes up not only in the human case, where reflection typically involves thinking about the intensional (meaningful) content of other thoughts and epistemic states, but also in computational contexts (see for example ·86/1 and ·108/1).

In many respects the architecture presented in this dissertation and embodied in 3Lisp can be considered to provide, at least in the first instance, only two kinds of referential access: *extensional* and *hyper-intensional*. The only entities that can be named or referred to, in 3Lisp, are: (i) entities that other structures or expressions name or refer to—the default case, in which one refers to them by using those other structures or expressions "transparently," in an extensional context; and (ii) other structures or expressions *themselves*, which is accomplished by using quotation (either «'» or «"…"») or the explicit NAME operator ('↑'),[4] thereby obtaining both referential (φ) and causal (ψ) access to that entity as a causally-efficacious mechanical ingredient.[5]

Even though reference to intensions is not supported in 3Lisp, however,[6] that does not mean that intensional issues are off the

1. References are in the form *page/paragraph/line*; with ranges (of any type) indicated as *x:y*. For further details see the explanation on p.·
2. «Refer to the "three spellings of intentionality" sidebar—but where is that?
3. Included here as ch. 3c.
4. See 115/-2/1:2.
5. Except cf. A66, below.
6. That is: the ability, given any expression or structure X, to construct a different expression or structure Y, such that the *extension* of Y is the *intension* of X.

table—as indeed they cannot be in any functioning computational system. As (rather verbosely) discussed in dissertation §4c.i (included here as ch. 3c) the foundational notion of λ-abstraction, and the behaviour associated with the primitive closure bound to the name LAMBDA in the global environment, can only be understood in intensional terms.

As explained there, the "function" of a term of the form '(LAMBDA *PATTERN BODY*),' in informal terms, is in some sense to "capture" both the declarative and procedural intension of the expression *BODY*, in such a way as to allow that intension to be used or invoked in other contexts. The issue is that the intensions of expressions are in general context-dependent. Lacking a theory of intensions, I was not able to provide 2Lisp and 3Lisp with a mechanism with which to name, denote, or refer to the intension of *BODY*. What reduction of the composite expression (LAMBDA *PATTERN BODY*) does, however, is to produce a structure (a closure) that is co-intensional with (i.e., has the same intension as) *BODY*, but that is also context-*independent*, so that that structure can be used, as appropriate, in any other context in which the intension of *BODY* might be needed.

No matter how useful, however, a mechanism that allows one to construct co-intensional expressions is still a far cry from being able to designate such intensions explicitly. The latter was an explicit design goal for Mantiq, as described in the Cover, for which 3Lisp was intended to be a design exercise. As mentioned there, moreover (cf. also annotation A3 of ch. 3a), one of the ways I imagined accomplishing this, in Mantiq, was by defining the structural field sufficiently abstractly so as to be able to *fuse* structural identity with an otherwise-motivated notion of intension or meaning (so that testing whether two structures were structurally the same could stand in for—could serve as the subpersonal correlate for[7]—whether they meant the same thing). By defining the structural field sufficiently abstractly, that is, en route to what I conceived as a "field-theoretic" view of computation, I hoped at least to reduce, and perhaps ultimately even dismantle, the distinction between intensional and hyper-intensional reference.

Even at the time, I viewed traditional model-theoretic accounts of meanings in terms of functions from possible worlds to truth-

7. See A24, below.

values as too coarse-grained for semantical as well as structural reasons—insufficiently articulated as regards issue of deixis and context-dependent reference, etc., as well as raising computationally intractable issues of identity, and therefore inappropriate candidates for fusion.

I still want to explore this design idea of adopting a "field-theoretic" approach to the structural field of a computational process, that honours intensionally-motivated constraints—one that, at least if implemented on currently recognisable hardware, would need to be backed by sophisticated relaxation algorithms to compute these more motivated but less fine-grained notions of structural identity. The situation is hugely complicated, however, by the fact that I no longer believe that "*the* intension" of an expression or structure is a well-defined notion. At the very least, the declarative "route" (φ) from sign to signified needs to be considered as much more continuous, or at least as a path with arbitrarily many distinct "points" along it, in order to deal appropriately with the complex sorts of deictic and indexical dependence that natural language illustrates and that a system like Mantiq would require. An adequate exploration of these issues therefore remains for future work.

A2 ·31/-1 While the first two issues (formulating an epistemologically adequate descriptive language and unifying that with a theory of computation or procedural consequence) were not addressed in 3Lisp, as described in the Cover it was my intent to address both of them in Mantiq.

A3 ·31//-2:-1 For more discussion of the dual-calculus nature of Prolog see ·50/1:21/1.

A4 ·32/1/-3:-2 At this point, in describing "a program able to reason about and affect its own interpretation," I have not yet distinguished the various meanings of the term 'interpretation' that will be discriminated later in the chapter (in particular, descriptive vs. procedural meanings), but in this context what I primarily had in mind is the more computationally prominent procedural notion of interpretation as *program execution*, rather than the representational or declarative sense familiar from logic and philosophical semantics.

A5 ·33/1/-4 As mentioned in annotation A2 of the dissertation preliminaries (ch. 3a), to minimise confusion I explicitly flag chapter references that

refer to chapters in the dissertation, of which only chapter 1 is included in this Volume,[8] so as to distinguish them from references to chapters in the present volume.

A6 :33/-1/-3 As remarked in the Cover to this chapter, the writing in the dissertation is rather confused about whether the declarative or representational interpretation of computational states had to be *externally attributed*—by, as it is said, we "external observers"—or whether that was only a contingent fact about current systems, with the possibility remaining open of computer systems achieving their own genuine or authentic *original intentionality*.[9] This and a number of other passages (e.g., see :35/1/9:10) are written as if such semantics *had* to be attributed—an empirically reasonable enough point of view in 1981 (when the dissertation was written), given the state of existing computer systems, and the philosophical strength of the "formal symbol manipulation" construal of computation. But even though I did not feel that most practicioners in artificial intelligence grasped the gravity of what original intentionality would require (full-blown normativity, essentially), I was nevertheless already disquiet about the adequacy of the formal symbol manipulation thesis. As a result, therefore, as pointed out in other places (e.g., see :44/1 and A24) it was not my intent to take as definite stand on the issue as these passages suggest. (Cf. also such passages as :35/1/9, where I suggest that some declarative interpretation was *tacitly* attributed—a separate and also equally indefensible claim.)

A7 :34/0/2:5 This comment that 2Lisp "makes explicit much of the understanding of Lisp that tacitly organises most programmers' understanding" would have stronger if it had made the normativity more central—e.g., by saying that 2Lisp makes explicit how it is that its processing regimen *honours programmers' attribution of representational meaning* (just as, in a sound inference regimen, the derivability relation honours the pre-theoretic attribution of referential meaning—cf. §5c of the Introduction).

A8 :34/1/4 See A3 (re :39/n3/3:6) in "Reflection and Semantics in Lisp," chapter 4 «decide where to have original»

A9 :35/1/9:10 See A6, above.

A10 :35/1/-7 It was misleading, and strictly incorrect, to suggest that declarative

8. For reference to an internet-accessible version of the entire dissertation see p. of the Introduction.

9. Cf. the discussion at :33/3.

import and procedural consequence be *independently* formulated. My overall intent was for declarative import to precede procedural consequence, both ontologically and explanatorily (modulo issues about dynamicism, grounds for pragmatist epistemology, etc.; see the Introduction p. ·, and A·). The idea was that procedural consequence—what happens to program fragments, how they are executed—would then be defined so as to honour that declarative import, just as derivation is defined in a sound logic to honour semantic interpretation. Theoretically, the procedural consequence (execution) could be defined entirely arbitrarily, but in practice not only would that not be the intent, but it would vitiate both the whole point and the intelligibility of the resulting system.

A11 ·35/1/-5:-1 A number of reasons led my to call both dimensions *semantic*. Superficially, it was rhetorically important since, as mentioned at numerous points throughout this volume, computer science uses the term 'semantics' for the procedural dimension—based in part on its adoption of a specificational view of programs, with the consequence that an "interpreter" is taken in computer science to be an engine that effects the behaviour that a program is taken to specify. More substantially, though, in spite of giving representational semantics or declarative import both ontological and explanatory priority (cf. the preceding annotation), as mentioned in §3 of the Introduction I was nevertheless deeply respectful of the overall dynamicism about significance in general that computational experience pushes towards, and hence sympathetic to a pragmatist orientation towards both reasoning and ontology. As will emerge in §·, in characterising 3Lisp I ultimately formulate a single overarching account of a computational process's significance, of which the declarative and procedural end up being aspects or projections, reciprocally tied together both structurally (causally) and normatively.

A12 ·35/2/7:8 By "recursively-specifiable but not compositional" I was referring to what is discussed in §6 of the Introduction. By "strictly compositional" I intended what I discuss there as the originating idea behind compositionality: that that is assigned as the "meaning" of an ingredient structure remains true to what, in at least some intuitive sense, the term, in a contextually-dependent way, means or refers to in that context, rather than to a full account of the contextual-

dependence of that meaning or reference (the latter being what the "recursively-specifiable" algorithm needs to work with).

A13 :36/1/9 In using the predicate 'abstract' to characterise internal or ingredietnal structures (the second analytic axis along which computational calculi are described), I did not mean to suggest that such structures are Platonic or otherwise immaterially diaphanous, but merely that the individuation of such structural elements can be, and in the 2/3Lisp case definitely is, "more abstract," in the sense of making fewer distinctions, than are neeeded to make sense of (the concrete materialities of) written textual or notation expressions. However the term "internal structure" (rather than "abstract structure" would have been more consonant with the ensuing discussion—e.g., where I say "each structural class be treated in a uniform way by the primitive processor."

A14 :36/-1:7/0 Although the dissertation asserts, in numerous places, that category alignment is an important aesthetic underlying the design of 2/3Lisp, the reason for that importance is not well explicated. There is a little discussion here, mostly about how failures to be category aligned tend to lead systems to resort to metasyntactic and metastructural access; see also the more extensive discussion at the end of §1.f.i, on pp. ·111:·112. Nevertheless, after the first reports on 3Lisp were published, I was struck by the fact that the idea of category alignment was treated by most people I spoke with as at least extraneous, perhaps a distracting red herring, and at worst theoretically inelegant.

As usual, I believe the five things: (i) that the critical response has some merit; but (ii) unfortunately, perhaps because of that merit, the original proposal was too quickly dismissed or ignored, at the cost of understanding either of the underlying issue or of what the proposed solution was aiming to—and to some extent did—accomplish; (iii) that the concerns raised in the criticism are conceptually orthogonal to the merits of what was being proposed; and (iv) that as a result it should be—indeed is—possible to develop a solution that does justice to both the proposal and the critique; but (v) that, perhaps surprisingly, the development of a solution that simultaneously honours the original insight but avoids the issues raised in the critique will be much harder than one might initially suspect, requir-

ing radical revisions in our overarching metaphysical and ontological frameworks.

What is right about the critique, to pick up the first of these points, is that it is endemic to programming practice to define more complex or abstract data structures out of simpler ones. Tying instances of such structures to the type structure of their implementation code not only fails to matter much, but contravenes some of the most important mandates of structured programming—that one not make one's code excessively implementation-dependent. Clearly, this aesthetic is deeply recognised in class-and object-oriented languages, and is embodied in the notion of an *abstract* data type. Whatever it is that motivates the category alignment mandate, therefore, should clearly be framed in terms of user-defined classes or types, not in terms of the primitive data structures that implement them. (In fact this very issue is discussed in §:, on p. :.)

What is nevertheless right about category alignment, to turn to the second point—or at least right about the intuition that led me to propose it—are two things. The first, adduced in this paragraph (·36/-1:7/0) is that categorical confusion tends to lead to gratuitous use of semantic ascent, in the form of quotation and other practices fundamental to genuine reflection. This is the thrust of pp. ·111:112, cited above; they point out how, in Lisp's case, a failure of category alignment leads to excessive (formally necessary but conceptually unwarranted) uses of quotation and calls to the primitive function APPLY. As these examples suggest, even at this relatively superficial level, sans category alignment, reflection gets very confusing, very fast.[10]

The second consideration motivating category alignment cuts deeper. It is almost fundamental to reflective processes that they tend to deal with the structures over which they are operating—i.e., the structures at one level below, which their own variables and data structures and arguments etc. denote—in terms of those structures' procedural and declarative semantical categories. Sometimes that is not true; sometimes there is something *particular* about an object-level structure or situation that requires dedicated focused reflective attention. But it is in the nature of things that it is typically in terms

10. Cf. my comment «where?», in discussing the differences between 2/3Lisp and Scheme, that, en route to reflection, quotation needs first to be understood, then to be disciplined, and finally to be unleashed.

of the (more or less reified[11]) categorical structure of the object level domain that reflective procedures are most commonly defined.

Because of the formality condition, however—or anyway whatever it is that is right about the formality condition, whatever the formality condition turns into on a participatory construal, etc.—all that the reflective process has effective access to is structure, not semantics. And so being able to define *structural* type predicates that sort structures according to their *semantical* character is critically important.[12] In the data-structure-oriented Lisp environment category or type alignment was a simple way of doing this. Once support for data abstraction is explicitly introduced, as for example it is in class systems and most object-oriented languages, more complex versions of such "alignment" would be simpler to provide.

As mentioned in the Cover, the issue is that declarative semantics (φ) does not cross implementation boundaries—*including the "implementation boundary" separating an abstract data type or class from the structures and operations that implement it*. That implies that, in a system of such sort, when a programmer defined a category or class, they would not only need, as at present, to define its *behaviour*, but would also have to specify its *declarative import* or *representational semantics*, or at least say enough about it to ensure that the programmer-specified behaviour *honoured the appropriate norms*.[13] And to do that would require adversion to a fully adequate ontological theory of the subject or task domain of the program itself, rather than merely the subject or task domain of the language processor. I.e., it would require, or involve, or however one wants to put it, developing an account of *program semantics*, not merely of *programming language semantics*.

There is no doubt that is a task that should be taken on. That was the task I intended to address in 4Lisp, the envisaged next step in the series of design studies en route to Mantiq. But as stated briefly in chapter 1 ("The Foundations of Computing"), the ontological problems proved daunting. The fact that I was not easily able to surmount them is why 4Lisp was never developed, and why Mantiq

11. That is one of reflection's advantages, that it can reify what is implicit one level below. See A75, below.

12. I once thought of this property as that of being *syncategorematic*, on the mistaken view that 'syncategorematic' mean *syntactic* (i.e., structural) *in virtue of categories*—or perhaps more simply, *syntactically (or structurally) categorical*. I still somewhat rue the fact that that was an egregious back-formation.

13. Cf. the dicsussion of Amala in the Cover.

has yet to be designed. It was in part to address them that I wrote *On the Origin of Objects*, which from this perspective can be viewed as an attempt to discern a metaphysical framework that could be serviceable for a task of this sort. Developing the insights articulated there into a theoretical system that could form the basis of computational analysis, design and construction is the task towards which the design of the fan-calculus[14] is oriented.

My belief remains strong that the resulting system would be powerful, useful, and elegant—as does my resolve to design it. Thirty years on from designing 3Lisp, however, I am not sure whether I can honestly yet say that I am more than halfway there.

A15 :37/1/8 Re 'independent': cf. A10, above, on passage :35/1/-7.

A16 :37/2/-5:-1 It would have been simple, and for some readers helpful, to frame some of these points in terms of the philosophical concepts of *semantic ascent* and *descent*. Thus in characterising the 2Lisp processor as semantically flat I am saying that, in 2Lisp and 3Lisp, normalisation (the default processing regimen) does not, but reflection does, engage in semantic ascent and descent.

A17 :38/0/-3:-2 The distinction between an object language and a meta-language is invariably context-relative. Languages, including small fragments and/or individual expressions, do not intrinsically have the status of being at the object or meta level; they acquire any such status only in relation to another language or expression or intentional system. In simple contexts the meaning may be clear, but complexities invariably arise in any context in which reflection, implementation, theoretical description, and semantic ascent (and descent) are all simultaneously at issue. In 3Lisp, for example, code at each level in the tower is simultaneously object language from the point of view of the level above, and meta-language from the perspective of the level below.

In the passage in the text, however, the distinction being made is not between levels, but beween whole systems. In particular, by 'meta-language' I am referring to the external descriptive language that a theorist might use to describe or theorise 2Lisp or 3Lisp—paradigmatically, the mathematical λ-calculus. By 'object language' I mostly meant 2Lisp or 3Lisp—though in the case at issue, regarding the designation of environments and continuations, the object

14. Cf. §· of the Cover.

language would be specific passages of 2Lisp or 3Lisp that were nevertheless "meta" to some other code or processing that those passages were engendering or representing (at a reflective level, in code for a meta-circular processor, etc.).

A18 ·38/0/-2 Note that it is *designators* of environments and continuations that are part of the protocol code. There is a strong sense, because of the existence of these designators, that environments and continuations are *themselves* part of the definition of 3Lisp, but that is not strictly correct. Better would be to say something along the following lines: that environments and continuations are extraordinarily good (realistically: indispensible) theoretical entities in terms of which 3Lisp can be found intelligible. However the truth of that statement should be not taken as implying that environment *structures* and continuation *structures* are a primitive part of 3Lisp. To speak in that way would be not only to confuse implementation with implemented, but also to fail to appreciate the importance of the declarative dimension of 3Lisp (and 2Lisp) semantics.

A19 ·38/1/-2:-1 This characterisation in terms of a limiting ideal is what in §1·e·iv I call a 'tower' view of 3Lisp; see especially ·101/1.

A20 ·40/0/-3:-1 I firmly believe the final statement in this paragraph: that nothing stands in the way of procedurally reflective, semantically rationalised versions of languages that support data abstraction, user-defined classes, and message passing. In spite of the promise made in the middle of the paragraph, however, the submitted dissertation did not show in detail how this could be done. As discussed in the Cover, and at length in A14 (re ·36/-1:7/0), above, the theoretical challenges are considerable.

A21 ·41/2/-1 Dismantling the distinction between declarative and procedural calculi (the second design goal identified on the opening page of the chatper) is of course one of the goal of Mantiq—in effect the subject matter of this entire paragraph.

A22 ·41/-1/-2 Cf. A11 (·35/1/-5:-1) and the discussion in the Cover, about the use of the term 'semantic' to characterise the unification of these aspects.

A23 ·43/-1/8 «Reference Dennett, Haugeland, Searle, as appropriate…»

A24 ·44/1/-3:-1 As stated in A6, above (at ·33/-1/3), the writing is less than clear about whether the declarative semantics *must* be attributed, or could potentially be original. *(cont'd)*

More important here, however, is the following point: because reflection is defined in terms of φ, the question of whether or not a system is or is not a reflective system depends on what one takes to be the status of the declarative semantics. It is not a peculiarity of the 3Lisp approach to provide reflection in an architecture defined in terms of a double (ψ/φ) semantical account, in other words. Rather, the notion of reflection—of a system reasoning or engaging in process that is *about* its own operations and structures—requires a prior, non-procedural notion of *aboutness*. Cf. the last sentence in the subsequent paragraph, which talks of systems dealing with their own ingredient structures and operations *as explicit subject matters*.

Hence my sense that, no matter how otherwise gracious, Friedman entirely misses the point in attempting to define a notion of "reflection without the metaphysics" «ref» (though cf. also A:)

A25 :44/-1:16/0 These two paragraphs are wordy and confusing. See the next annotation (A26).

A26 :46/0 It would helped, rhetorically, if these two paragraphs (:44/-1:16/0) had been phrased in terms of the personal/subpersonal distinction (a framing I of which I was unaware at the time).[15] Even then, though, the issues are subtle.

If, *qua* person, I think about Virginia Falls, then according to the Knowledge Representation Hypothesis (KRH) that happens in virtue of my brain's constituting of a subpersonal process P (formally) manipulating equally subpersonal representations of Virginia Falls. My personal-level *thoughts* about Virginia Falls, therefore, have, as their subpersonal correlates, something like *subpersonal symbolic representations of Virginia Falls*—i.e, subpersonal interior symbols denoting the falls. If therefore, at the personal level, I then *reflectively think about my (personal) thoughts about Virginia Falls*, then by the KRH that must happen in virtue of, at the subpersonal level, an internal processor P *manipulating internal symbols representing those thoughts*.

Now at this point the 3Lisp architecture makes an assumption, which I will call ρ, that it is important to spell out. Depending on one's viewpoint, one might either characterise ρ as so obvious as barely to deserve mention, or indict it as a devious sleight of hand. Call my personal-level thoughts about Virginia Falls T_1, and their subpersonal correlates R_1. Similarly, call my reflective thoughts

15. «Reference: Dennett—was he first?»

about my thoughts about Virginia Falls T_2, and their subpersonal correlates R_2. What assumption ρ has to do with is the declarative semantics of R_2.

According to the KRH, R_2 should be an internal representation of T_1. Instead, what the 3Lisp architecture presumes is that we can treat R_2 as—can assume it is equivalent to, can in some sense take it to *be*—a representation of R_1. That is, we can define ρ as follows:

ρ Instead of taking the subpersonal correlate of a reflective thought to represent a personal object-level[16] thought, we instead take it to represent the *subpersonal correlate* of that object-level thought.

Assumption ρ underwrites the semantic interpretations of reflective structures presented in the rest of the dissertation. By stipulation: (i) $\varphi(T_1)$=Virginia Falls themselves, the cascading sheets of water; and (ii) $\varphi(T_2) = T_1$, my falls-directed personal-level thoughts. According to the KRH, $\varphi(R_1)$ is the same as $\varphi(T_1)$—that is, the same cascading sheets of water.[17] T_1 and R_1 are distinguished by *type* or *form*, that is, not by *content*. The former are thoughts, the latter are (formal) representations, and both designate the same thing—namely, falls. At the meta or reflective level, however, the preconditions arise for conceit ρ. By the KRH, $\varphi(T_2)$ and $\varphi(R_2)$ should again coincide; they should both be T_1. By ρ, however—and thus what is embodied in the 3Lisp architecture—instead of having $\varphi(R_2)=T_1$, we have $\varphi(R_2)=R_1$.

This way of looking at things suggests that ρ is a cheat. But there is another way to understand ρ, according to which ρ is not only far less problematic, but actually well-motivated. Instead of taking subpersonal meta-level representations to represent other subpersonal symbols (instead of representing the personal-level thoughts those symbols are correlated with), one could instead say that really, as required be the KRH, they *do* represent the personal level thoughts, but do so in virtue of bearing a closely-allied but nevertheless distinct relationship, pretty much like designation, to the corresponding subpersonal correlates. Label that "closely-allied but nevertheless distinct relationship" φ′. Then, using the example above: if we take $\varphi(R_2)$ to be T_2, we would have the following:

16. Cf. A15, above.

17. That is what it is to say that personal-level thinking is constituted by a subpersonal processor manipulating internal *representations*.

1.	$\varphi(T_1)$ = Virginia Falls	— stipulation
2.	$\varphi(R_1)$ = Virginia Falls	— (1) plus KRH
3.	$\varphi(T_2) = T_1$	— stipulation
4.	$\varphi(R_2) = T_1$	— (3) plus KRH
5.	$\varphi'(R_2) = R_1$	— proposal

That is, as these equations make clear, φ' would in a sense be the *subpersonal correlate of designation* (φ).

What is haunting about this admittedly arcane discussion is that in teasing apart distinctions (e.g., between personal-level thoughts and their subpersonal correlates), and then following up the logical implications of so doing, we end up teasing apart other things (designation and its subpersonal correlate), in a process that is reminiscent of the very problems that 3Lisp inadvertently introduced, by being a stickler for use/mention distinctions. That is: whereas 3Lisp was rigorous about distinguishing signs from what they signify, to the point of ultimately becoming unusably fastidious, the present discussion is doing the same thing as regards the personal/subpersonal distinction.

I believe the morals are profound. As argued in "The Correspondence Continuum" (ch. 10), computational systems are permeated with cascades of relations between and among entities or relations that for some purposes can be seen as sufficiently similar so as not to warrant making a distinction between them, and for other (perhaps rare) purposes distinguishable (e.g., φ and φ', in the case at hand). To put it in the terminology of *On the Origin of Objects*,[18] for some purposes it may be important to register a distinction between a thought and its subpersonal correlate; for other purposes, not. The challenge is to frame the background metaphysical/ontological/epistemological assumptions, and the working theoretical framework, in ways that can honour this approach of doing justice to distinction in a (normatively-driven) context-dependent way. The former, of course, is the aim of 03; the latter, of the fan calculus project suggested in the Introduction.

A27 :46/1 This paragraph, too,[19] would have been more clearly explicated in terms of a personal/subpersonal distinction. At issue is what subpersonal activity underwrites or is correlated with a personal-level

18. See also "Representation and Registration," ch. · of Volume ii.

19. Cf. the former annotation, A26.

reflection. On the surface, the claim is this: it does not consisting of interior process P *reflecting* on R_1, which would seem circular. Rather, the architectural idea is that P considers R_2.

What is striking is that this claim is framed with reference to the level-shifting view.[20] The story on the tower view is more interesting. In particular, on the tower view, one could say that, sure enough, when, at the personal level, someone has a reflective thought T_2 about base-level thought T_1, that happens, subpersonally, in virtue of the person's interior (subpersonal) process P_1 *reflecting* on subpersonal correlate R_1. But then we discharge the threat of circularity by offering an account of what it is for subpersonal process P_1 to reflect on R_1. Specifically, we iteratively apply the KRH, and say that P_1's reflecting is "sub-subpersonally" realised in virtue of P_2 manipulating R_2.

While it has real merit, positing such an iterative application of the KRH is also somewhat ironic. It brings up the point made «where? at least point back to §6 of the Intro. also ·105/1?» that one virtue of substantial architectures is that some questions need not be answered. But it ties in as well to the point made in the previous annotation; that the very distinction between the level-shifting and tower views is ultimately a choice between two behaviourally-equivalent registrations.

Perhaps the most important moral, in this case, is that the rough equivalence of these two views begins to undermine the integrity or anyway absoluteness of the personal/subpersonal distinction itself—a point towards which many other issues militate as well.

A28 ·46/2/-3:-1 Cf. "Varieties of Self-Reference" (ch. 6)

«Needs work. Cf. comments in the POPL paper. Also see earlier annotation, and comments at the beginning, about the relationship between self-reference, introspection, and reflection…»

A29 :47/2/8 Snobol ("String Oriented Symbolic Language"), a string-processing language developed at AT&T Bell Laboratories in the 1960s, had the distinctive property of allowing strings to be treated as programs, thereby enabling programs to be dynamically constructed and executed on the fly. Famous for treating patterns as a first-class data type, Snobol served in some ways as a precursor to such modern scripting and text-oriented languages as Perl.

20. For a discussion of the level-shifting and tower views, cf. §1·e·iv, especially ·101/1.

A30 :52/-2/-4 This is just one of several places throughout the chapter[21] where I talk about reflection needing "actually to matter" to the process in which it occurs. Unfortunately, the sense that I had in mind, or at least that I could make good on at the time, was more one of *having concrete physical consequence* than of *being important*. I was perfectly well that a more encompassing normative sense of mattering was fundamental to the long-term goal of genuine reflection.[22] However, as discussed at numerous places in this chapter (e.g., see :43/-1), I was under no illusion that 3Lisp achieved the requisite semantic originality that genuine mattering would require. For these reasons, it would have been better to have written more modestly of something like *effective consequence*.

A31 :54/1/9:10 The ability to plant seeds, now, so as to ensure future reflection is tremendously important—not only to the design of 3Lisp and other procedurally reflective systems, but for an understanding of reflection in the general case. It is certainly a staple of mundane personal psychology: to be able, in advance, to "set oneself up" so as to ensure that later, when some circumstance arises, one will at that point stop and reflect (e.g., about what to do or not do).

A32 :56/2/4:8 The claim that no computational process can achieve the limit of reflexive (as opposed to reflective) thought is something I thought at the time the dissertation was written (1981), and so I have left it standing. However, it is not a statement I would entirely endorse today (2012). Increasingly, I have come to believe that there *are* ways in which it is possible to have a "I am now thinking" thought refer to itself, or perhaps more accurately to *include itself within its referential domain*, without invoking a Necker-cube like reverberation between one state and another—perhaps in something like the way in which non-well-founded set theory[23] supports the notion of a set having itself as a member. Descartes' *cogito*[24] does not seem semantically ill-formed, after all, even it feels a little phenomenologically unstable—though the shifting-back-and-forth that accompanies thinking about it may derive from a double self-reference, involving not only (reflexively) thinking that one is thinking, but also (reflectively) thinking—or recognising—that one is in fact doing that. *(cont'd)*

21. See :59/0/10, :62/0/4, :62/-1, and :104/1/-3:-1.
22. See e.g. Haugeland's "Truth and Rule Following" «ref».
23. «Ref Axcel, others?»
24. Independent of the 'ergo sum' part.

The epistemic achievement necessary to genuine reflexion, I believe, would involve entertaining a reflexive thought *quietly*, as it were—to hold a self-encompassing thought, in such a way that awareness of its reflexive self-referentiality (or semantic self-inclusion) does not lead one into a kind of vibrating or alternating epistemic state. While not necessarily easy, I believe not only that this can be accomplished, but also—perhaps oddly, perhaps not—that doing so relates to a number of forms of self-referential discipline that have been developed in various Asian and other meditative and mystical traditions. At a more mundane level, and apparently unlike some others, I do not believe that either the meaning or the truth of such statements as that "all statements are perspectival" need in any way be undermined by the fact that they apply, among other things, to themselves.

As explained in more detail in "Varieties of Self-Reference" (ch. 6) «check», I characterise as 'reflexive' not only those states, processes, expressions, etc., that are strictly *self-referential*, in the sense of being their own semantic extension (e.g., "this very five word phrase"), but also those that we might call *self-applicable*, in the sense of including themselves within their referential or semantic extension (such as "all phrases in English"). By 'reflective,' in contrast, as here, I refer to processes of "stepping back" and assaying, from a distinct vantage point (cf. A·), another part or aspect or period of oneself. There is no doubt, according to this distinction, that even if computational models of reflexion are possible, 3Lisp was correctly described as a model of computational reflection. (Cf. also §1·b·iv, starting on p. ·59.)

A33 ·58/2 Cf. the previous annotation (A32, ·56/2/4:8). As always, "stepping back" must be defined with reference to an encompassing frame of declarative or representational semantics (φ), and hence depends on a background structure of norms. While it is true, as mentioned in the previous paragraph in the text, that one cannot step outside of, for example, language, that does not imply that one cannot achieve a normatively governed vantage point from which to regard language with some detachment. It is just that that detachment will not be complete.

A34 ·62/0/4:6 Cf. A30, above (·52/-2/-4). As is evident here, the reason for the inter-

nality of the causal relationship has more directly to do with effective procedural consequence than with anything authentically normative (though of course the former is required in order to honour the latter).

A35 ·63/-2/7:8 At the time this was written, I was already starting to reject the claim that computational processes are *formal*, in the sense of operating independently of their semantic interpretation, but I had yet to question another widespread assumption: that computational arrangements are *abstract*—or anyway, as is being said here, temporal but not otherwise physical. I have come to profoundly disagree with this view, believing that computational processes are as much denizens of the material or physical world as, for example, are we. See both 03 and A05.

A36 ·64/1/7:8 «Reference the discussions in other papers—POPL? Prologue? CC? I forget where this is talked about, complete with those α/β figures, etc.»

A37 ·66/1/3:4 In the dissertation I called these *interpretive* and *communicative* reductions, respectively, but in retrospect that choice of terminology seems ill-advised. For one thing, to employ the adjective 'interpretive' for those processes that contain a single interior locus of agency is too committed to the computer science sense of the term 'interpreter,' which I have explicitly set aside in favour of 'processor.' Similarly, to employ 'communicative' for interacting multiple interior loci of agency commits to their interaction being one of "communication"—i.e., to involve the exchange of meaningful entities, which is a more specific claim about the nature of process-process interaction that I want to be committed to at this level.[25] The predicates "serial" and "parallel" seem both simpler and more consonant with general computational practice. Because this terminological change reflects a substantial intervention, however, I have marked the uses of both terms with brackets here and through the rest of the chapter.

A38 ·66/1/6 In the dissertation this was written "how these processes communicate" (emphasis added); I have changed it to 'interact' in line with the previous annotation (A37).

25. By analogy, cf. John Haugeland's characterisation of digitality («ref») in terms of 'reading' and 'writing'—a similarly unwarranted use of intentional vocabulary to characterise forms of interaction that are not necessarily intentional at all.

A39 ·66/1/9 This is the English rather than computer science sense of 'interpreted'; I would now say "registered."

A40 ·67/1/-6 Cf. also ... «Point also to other papers and commentaries as appropriate»

A41 ·67/-1/4:6 I continue to believe that the relation between *programs* and programming *languages* is of far more theoretical importance than is normally recognised. See «...» for a discussion of the relation between programming language semantics and program semantics. «...»

A42 ·71/0/2:3 The last sentence in this ¶ is too strong. I had not yet developed the language of *registration* (cf. 03 and "Representation and Registration", ch. · of Volume II). In its terms, I would rephrase the first sentence of the paragraph as follows: "To implement Lisp, in other words, all that is required is the provision of a process that can be registered as consisting of the Lisp structural field and the interior Lisp processor." So stated, the claim would be so obvious as not to have been worth making. The point is that the dissertation was written in the grip of an untenably naïve realism, which occasionally (e.g., see A45, below) required caveats and maneuvering to sidestep.

A43 ·72/1/2:3 The characterisations of interpreters (in the computer science sense of the word), compilers, etc., given over the previous several pages, are all framed behaviourally, rather than in terms of declarative or representational semantics. And so it might seem as if a mathematical theorisation would require formalising only ψ, not φ or a more generalised significance function Σ. But as discussions throughout the chapter make evident, I believe that no analysis that does not treat φ and declarative import generally would cut to the heart of the computational phenomenon.

A44 ·72/3/7 As reported on Wikipedia,[26] the first evidence of a COME-FROM instruction appeared under the label 'CMFRM' in humorous lists of fraudulent assembly language instructions. It was elaborated upon in Clark, R. Lawrence, "We don't know where to GOTO if we don't know where we've COME FROM", *Datamation*, 1973, written in response to Edsger Dijkstra's "Go To Statement Considered Harmful" «ref».

A45 ·73/1 This and the subsequent paragraph are clearly an informal and not especially clear amalgam of Fodor's *formality condition*, Dennett's *intentional stance*, and a distinction between *original* (*authentic)*

26. «Ref»

and *attributed* (*derived*) intentionality. Fodor's classic formulation of the formality condition appeared in 1981, the year this dissertation was written;[27] Dennett's *Intentional Stance* was not published until six years later (Dennett 1987), though formulations had appeared earlier.[28]

For numerous reasons I no longer believe that 'computational' is best understood a predicate on explanations. My current sense is that the best way to understand the reason I thought so then is that in 1981 I was still in the grip of an excessively naïve realism—or anyway did not yet have vocabulary in terms of which to say anything different (cf. A42, above). The language of registration (03) would have allowed the points to be much more simply and effectively presented.

Note, moreover, even setting aside issues of theoretical vocabulary, that the thesis that computational semantics is necessarily attributed or derivative (see A6, above) does not imply that 'computational' is a predicate on explanations. As I have said elsewhere,[29] to say that intentionality is derivative is not to say that it is not *real*; it is real *as derivative*. A theory of derived intentionality can still be perfectly ontological; it would just need to explain the ontological conditions *for the interpretation being attributed*. (In terms of metaphysical status, all that derivative intentionality denies is that the intentionality is *intrinsic*—but that is a different thing. It would be an stringently impoverished metaphysics, to say the least, that restricted reality to the intrinsic.)

The main point in the text, however, is that the fundamental thesis argued in the dissertation—that reflection is straightforward to understand and implement if built on a semantically clear base—implies that developing an account of computational reflection, and hence designing 3Lisp and like languages, requires not only understanding such philosophical views about the nature of computing, but effectively "building them in" to the resulting reflective architecture.

A46 :73/-1/-3:-2 The term 'syntactically' is used here as in philosophy, not as in computer science. At best, in computer science one would say 'structurally,' but the meaning would be so deeply assumed that to give it a label at all would seem odd.

27. «Fodor 1981.»
28. «Check, and reference if appropriate»
29. «Where?»

A47 :75//-2:-1 As explained in the annotation «where?» to "Reflection and Semantics in Lisp" (ch. 4), at the time this dissertation was written, perhaps in part because the project emerged from several years working the area of knowledge representation, I was singularly focused on an *ingrediential* view of programs. I did not considered the position, much more commonly held in computer science, of viewing a program as a *specification of*, rather than as an *ingredient within*, a computational process.

A48 :76/-2/1:2 Philosophical readers will find it awkward to characterise both aspects as *semantic*. Cf. AII, above (:35/1/-5:-1).

A49 :77/2/-5:-4 «probably *Situations and Attitudes*, but maybe one of the earlier papers—check»

A50 :77/-1/-7:-5 It is because 'CAR' is being used as a term of English in the text, rather than as a Lisp identifier, that it is formatted in this paragraph (three times) and elsewhere throughout the chapter in a serif rather than sans-serif font—i.e., as 'CAR' rather than 'CAR.'

A51 :78/n12/1 The referenced postscript is contained in 3.f.iv (p. 246) of the full dissertation—not reproduced here, but to which a link is provided on p. :.

A52 :79/0/1:3 Cf. §5c of the Introduction, on p. :.

A53 :79/-1/-3:-1 The statement that I will consider cases where s_1=d_1, while strictly correct, is misleading. While it is true that self-reference of this sort is discussed from time to time, such as in §1·b·iv (pp. 59–63), the topic is raised mostly in order to contrast it with the sorts of self-reference with which the study of reflection is concerned.

A54 :80/0/1 Strictly speaking, my reply "Mark Twain" is presumably an *instance* or *use* of the referent of 'the pseudonym of Samuel Clemens,' on the assumption that the latter refers to a *type*—but the point is clear enough. In general, as pointed out in the Cover («ref»), the dissertation (fortunately[30]) pays little attention to type-token distinctions.

A55 :80/0/5 See chapter 8 of *On the Origin of Objects*, pp. 243-67, for an extended discussion of designation-preservation under the label "preservation of reference."

A56 :80/0/-5:-4 To claim that derivability (⊢) is designation-preserving is sloppy phraseology. What I meant was that, in the ordinary course of things, derivability is not a "level-crossing" operation. One can in-

30. 'Fortunately' because the traditional type/token distinction is profoundly too simplistic to deal appropriately with the range of one-many relations than permeate computational systems.

terpret the claim in a more mundane way as saying that if, from $\alpha_1 \ldots \alpha_k$ one were to derive β on the grounds that β is true if $\alpha_1 \ldots \alpha_k$ are true, then derivability has preserved the designation *truth*. But as well as being vapid, this is false; from falsehood one can derive anything, including claims that are true. But that was far from my intent.

A57 ·80/1/-7 "Crucially distinct" but of course normatively related; cf. §5c of the Introduction.

A58 ·83/-1/5 There is only one numeral per number within any given 2Lisp structural field; there is no need for multiple tokens or instances. So if there two distinct composite structures—such as for example (+ 2 3) and (if (= x y) 1 2)—the occurrences of '2' are structurally identical, rather than each having its own distinct "token" or "instance" of a common type. Think of the complex structures as manifolds that encapsulate the self-same unique numeral. That identity does not make the numeral into a type, of course; a different 2Lisp process, with a distinct structural field, would in some sense have its "own" numeral 2.

The more flexible forms of identity possible within structural fields makes the issue of types/tokens/instances/uses of types—that is, the question of what is *one* and what is *many*—spectacularly more complex in computational cases than in the familiar case of written, lexical expressions.

A59 ·84/1/-8 I put "operation" in quotes because of the claim that, *au fond*, quotation should be understood as a referential or naming convention, not as a procedure. As discussed in «where?», the operational consequence (ψ) of quotation should be defined, derivatively, terms of its referential function, rather than—as is so common in programming languages—being taken as a primitive behavioural operation.

A60 ·85/0/4:5 The λ-calculus does not provide for the definition of names. Functions can be designated only by use of the complex expressions that are needed to designate them. What I had in mind, when writing that we could "remove atomic designators," was that we could define a variant of Lisp that, like the λ-calculus, did not allow definitions, but instead required use of such full composite expressions in every function position. But the phrasing in the text—"the ability to *name* composite expressions as unities" (emphasis added)—is dou-

bly confusing and wrong. It is not that in a dialect that allows definitions that one names *composite expressions*; rather, one introduces unitary names to name what those composite expressions denote (i.e., functions). Secondly, the point is not that one names things as unities, but that names are ways to refer to (arbitrary) things with unities. What is unitary is the name, not the named. What I should have said is that, in the λ-calculus, one *denotes functions with composite expressions instead of with (instances) of unitary names*.

A61 ·86/1/1:2 The story would end only if one could then show that the procedural consequence (ψ) honoured the declarative import in a normatively appropriate way; see §5c of the Introduction.

A62 ·86/1/-7:-6 Cf. the discussion of intensionality (i.e., 'intensionality-with-an-s') in AI (·31/-1/5), and in dissertation section 4·c·i, includeded here as ch. 3.c.

A63 ·86/2/6:8 As is true in so many aspects of the design of 2Lisp and 3Lisp, there may seem to be a considerable discrepancy between the underlying philosophical motivation for this point (about procedural consequence affecting the context of declarative interpretation) and the almost triviality of the technical examples brought forward as illustration. In making this particular point, and more generally in reciprocally defining declarative import (φ) and procedural consequence (ψ) within an overarching general significance function (Σ), as described in the next paragraph, I meant to do justice to the dynamic, contextual dependences of reasoning and language in general—such as our needing, in rational thought, to keep up, in a fully participatory way, with a constantly evolving world, some of which changes are the result of our own doing. Examples would range from such simple examples as mundane temporal dependency (using 'yesterday,' tomorrow, to refer to what we today refer to with 'today') and performatives ("I promise to bring you coffee") to the sorts of consideration that underlie pragmatist epistemology in general. Causing a side-effect to a variable hardly connotes the richness of the phenomenon.

A64 ·88/-1/1:3 This is the equivalent, in a computational context, of saying something that would be obvious, logically: that one cannot specify a sound proof procedure (⊢) without first having in mind an interpretation function for it to honour.

A65 ·88/-1/3:7 This is too strongly stated. Full independence is not required; the two accounts could be reciprocally co-constituted. What I should have said is that defining a processing regimen in a calculus in which there is *nothing more* to meaning than "how the symbol or structure is treated" would not just evacuate the system of any semantic or intentional interest; it would also deprive it, in my view, of any claim to being a computational system at all—instead, reducing it to "naught but mere mechanism." Oil refineries, after all, are constructed of parts that have procedural consequence.[31] Computation, in my book, in spite of its abiding concern with mechanism and effectiveness, is nevertheless a fundamentally intentional phenomenon. (Cf. §5c of the Introduction, and AOS).

A66 ·99//1 An important property of reflective procedures implicit in this model was not adequately explained in the dissertation.

As will become increasingly evident as this chapter proceeds, and is spelled out in detail in the dissertation's remaining chapters, reflective procedures are used in 3Lisp in those cases that would involve the use of *intensional* procedures in non-reflective languages[32]—i.e., procedures that, in computational (ψ) terms, "do not evaluate their arguments," or, to put it more philosophically, procedures whose argument positions are opaque or intensional, rather than transparent or extensional. Quotation is a paradigmatic intensional operator, but others are ubiquitous, such as: desires, belief reports and other statements of epistemic state (such as memory reports) in natural language and thought; possibility and necessity operators, in logic; and LAMBDA, classical IF statements, "left-hand-side" expressions in assignment statements, etc., in computing. In 3Lisp, reflective procedures are used to subsume all such intensional practices.

One might expect it to follow that 3Lisp reflective procedures would not "evaluate" (i.e., normalise) their arguments—or again, to put it in philosophical terms, that the argument positions of reflec-

31. Someone might object that programs specify or represent the behaviour they result in, whereas mechanical parts, of the sort out of which oil refineries are built, simply have causal consequence. Perhaps that is so. The point is that, in order to make out a distinction like this between *specifying behaviour* and merely *leading to behaviour,* one needs an account of what it is to represent or specify (i.e., something like φ).

32. Including not just 2Lisp and all prior dialects of Lisp, but all programming languages other than 3Lisp.

tive procedures would be opaque or intensional. Interestingly, that is not so—at least not in any simple sense.

Note that line 18 of the RPP is the only place where reflective procedures are ever invoked (that is: where their arguments are bound, their closures expanded, etc.). And as figure 15 makes clear, in that context, they are invoked perfectly extensionally—their argument positions are perfectly "transparent." In fact from a certain perspective one can correctly say, of 3Lisp, that *all* procedure calls are extensional—that, ultimately, there are *no* opaque or intensional argument contexts at all.

What is going on is this. If, in the course of regular "user" or "object-level" code,[33] the processor encounters a redex of the form (PROC α_1 α_2 ... α_k), where PROC names a "reflective procedure"[34] or is bound to a reflective closure, then, before so much as glancing at the expressions in argument positions α_1 α_2 ... α_k, the processor effects a level-shift—"backing up," to use the language of the Prologue—so as to obtain an appropriately detached vantage point from which to consider the situation. In 3Lisp, there is exactly one such "appropriately detached vantage point": line 18 of the RPP. From that vantage point, the structures in argument positions α_1 α_2 ... α_k are then *referred to*, perfectly extensionally. Sure enough, to switch again to philosophical jargon, those structures are *mentioned*, not used. But—and this is the important point—*mention* is a perfectly valid form *of genuine extensional reference*. It is genuine extensional reference to the expressions or structures that are occupying argument positions α_1 α_2 ... α_k.

More generally, the 3Lisp architecture illustrates a non-standard approach towards opaque or intensional contexts. Traditionally, we assume that functions or operators that take their arguments in an opaque or intensional context work in the following way: (i) they treat their arguments differently from standard-issue (extensional) functions or operators, but (ii) they do so within the context of the interpretation of the sentences or complexes in which they occur. In 3Lisp, in contrast, a "reflective redex" in the sense just described[35] is understood as follows: (i) its occurrence signals a shift in interpre-

33. Cf. AI5 (re :38/0/-3:-2).

34. The reason for the quotation marks will be explained in a moment.

35. I.e., a redex of the form (PROC α_1 α_2 ... α_k), where PROC names a reflective procedure or (equivalently) is bound to a reflective closure.

tive context, to the reflective (meta) level, but (ii) in that different context the function or operator treats its arguments in the standard transparent way. In sum, rather than viewing opacity as a different kind of reference (reference is always extensional—that is how reference works), 3Lisp views it as a change in interpretive context.[36]

A67 ·101/1 This paragraph is the one place in the chapter where I have added entire sentences, in order to make a point clear. In the original dissertation, this paragraph consisted, in its entirety, of:

> "We will not take a principled view on which account—a single locus of agency stepping between levels, or an infinite hierarchy of simultaneous processors—is correct: they turn out, rather curiously, to be behaviourally equivalent. For certain purposes one is simpler, for others the other."

The terminology of 'level-shifting' of 'tower' views is discussed in §1·e·iv (see especially ·101/1), and also in "Reflection and Semantics in Lisp" (chapter 4) and "Implementation of Procedurally Reflective Languages" (chapter 5).

A68 ·106/1/-2 In the original dissertation, the following parenthetical comment was inserted at this point (following the words "and so forth"): "It is important to recognise that the suggestion of constructing a reflective variant of the λ-calculus represents a category error." A few years later, however, contrary to this statement, I did informally define a reflective version of the λ-calculus, as a vehicle in terms of which to explain reflection to Jon Barwise.[37] I have therefore omitted the parenthetical from this version.

The motivation for the parenthetical remark is clear enough from the surrounding text. At the time I viewed the λ-calculus as fundamentally a declarative language for denoting functions, not as a procedural calculus. But on reflection I am not sure that is entirely correct. α and β-reduction are deeply enmeshed in the definition of the λ-calculus, and although there is no requirement that terms be reduced, and the Church-Rosser theorem allows one to side-step issues of reduction order, and so forth, I believe that we are correct to call the λ-calculus a *calculus*, not simply a *language*. The system is

36. The example illustrates a general point made in the Introduction: of how philosophical insight that can be wrested from intensive engagement in the details of a computational system

37. Cf. annotation A41 in ch. 4, particularly note 10.

more implicitly procedural than the original passage suggested.

A69 ·108/0/2 «The following overlaps with annotation A13 in the POPL paper. Combine into one (the POPL version is better, except that I should include the "not pass functions upwards" comment from this one), and then simply have one annotation refer to the other.»

In a colloquium in the Artificial Intelligence Laboratory at SRI International, in the spring of 1982, I gave one of the very first talks on 3Lisp. As it happened, John McCarthy (inventor of Lisp, and designer of Lisp 1.5) attended. Though as a young student I was almost paralytically anxious about making this claim in front of the great master, I nevertheless proceeded with what I had planned to say, and claimed that, according to my analysis, traditional Lisp's dynamic scoping protocols were a "mistake," to which quotation and other metastructural manoeuvrings were a partial work-around—in particular providing a way of handing closures "downwards," though there was no way to pass them "upwards" (in terms of the usual notion of a control stack; this has nothing to do with the reflective hierarchy).

To my surprise—and considerable relief—John McCarthy very graciously agreed.

A70 ·113/0/1:2 More details are also given in "Reflection and Semantics in Lisp," included here as ch. 4.

A71 ·114/-4/-3 Though, as stated here, I originally defined closures to be pairs, by the time the POPL paper was written ("Reflection and Semantics in Lisp," ch. 4) I had given them their own distinct structural category. There is merit in both approaches.[38]

A72 ·119/1 While it is correct that lambda has first and foremost to do with naming, I do not believe that this paragraph is strictly correct. For a better analysis see dissertation section 4·c·i, included here as ch. 3.c.

A73 ·120/b2/-3:-1 «Ref Quine: from "On What There Is," *Review of Metaphysics*... ...;included in *From a Logical Point of View* (Harper & Row, New York: 1953)»

A74 ·121/1/-2:-1 Cf. A66 (re ·99//1), above.

A75 ·121/-1/-7:-5 This claim is extremely important: (i) that what was *used* prior to reflection is *mentioned* upon reflecting; (ii) what was *tacit* prior to reflection becomes *used* upon reflection. Not only did it influence

38. On the one hand, closures, like all other structural categories except atoms (variables) and pairs (redexes), are normal form. On the other hand, like atoms and pairs, and unlike the other normal form categories, closures are not unique designators.

the approach to real-world ontology that is sketched in 03; it also infected the ideas I was mulling on, at the time, about fusing higher-order and intensional "objectification" levels in Mantiq (cf. A1).

I still believe that a substantial issue remains lurking here, with which a proper theory of cognition should come to grips: relations between and among processes of

1. *Reification*—leading us to find the world intelligible in terms of objects;
2. *Semantic ascent*—generating quotation, meta-level concepts and expressions, and other forms of symbolic or cognitive "mention");
3. The use of *higher-order* structures (such as higher-order functions); and
4. *Reflection*—what we might call "*procedural ascent and descent*," involving all the issues adumbrated here, about stepping back, vantage point, etc.

In our formal efforts to be rigorously clear about the *differences* among these notions, we sometimes fail to recognise their *similarity*—and more seriously, what may be their common genealogy.

Note that Friedman and Wand's "Reification: Reflection Without Metaphysics," a paper I cite as indicative of the general reaction to 3Lisp, which set its semantical approach aside, can be understood in this light to be an attempt to wrestle with the first issue on this list without addressing the other three.

A76:127/-1/-9:-6 That declarative semantics does not cross implementation boundaries is an extraordinarily serious issue, which has yet to be theorised.

Suppose that architecture or virtual machine Y is implemented on top of language or system X. The question has to do with which of various properties P_i exemplified by X (the underlying system) are "inherited" by—i.e., true of—system Y, in virtue of the implementation relation holding between them. The answers are both complex and illuminating. There is no way that Y can be a "real-time" system, for example (in the sense of providing metric guarantees about certain kinds of behaviour, such as providing support for a routine to run exactly once per second), unless X is also real-time. So, to adopt a convenient way of speaking, I would say that being

real-time "crosses implementation boundaries downwards" (that is: that from a system's being real-time, one can conclude that the system on or in which it is implemented is also real-time—and hence all such systems below it, down to the hardware). Conversely, "being a finite state machine" is a property that crosses implementation boundaries *upwards*, since there is no way to implement a machine with an indefinitely unbounded store on top of one that has no such store. Needless to say, being a finite state machine does not cross implementation boundaries *downwards*; you can perfectly well implement a finite state machine in Lisp, which is not one.

The present point is that declarative semantical properties in general—and thus reflection in particular, since it is defined in terms of declarative semantics—*do not cross implementation boundaries in either direction*. From neither X nor Y's being reflective, in the above example, can one deduce anything about whether the other is reflective.

For further discussion see comments in § «?» of the Introduction, and AOS.

A77 :129/-3 As mentioned in the Cover, I believe it is fair to say that the hopes expressed in this paragraph were entirely in vain. Dan Friedman, of Indiana University, was one of the most enthusiastic proponents of reflection in the programming language community; I owe him a huge debt of gratitude for the enthusiasm and support he offered subsequent to the publication of the POPL paper introducing 3Lisp ("Reflection and Semantics in Lisp," included here as ch. 4). However as perhaps best illustrated in his own paper with Mitchell Wand,[39] the first thing that most people did, in bringing reflection into their own work, was to dismiss every one of these six claims.

For some of the reasons for this dismissal see the discussions both in the Cover and in the Introduction. Fundamentally, I believe that it stems from the confluence of two issues: (i) the lack of appropriately strong theoretical engagement discussed in §1 of the Introduction between and among philosophical enterprises (philosophical logic, philosophy of language, philosophy of mind), the formal representational tradition (mathematical logic, computer science data bases, the AI and knowledge representation communities, etc.), and the programming language community; and (ii) the inadequacy of our current theoretical frameworks.

39. "Reification: Reflection Without Metaphysics" (Friedman & Wand, 1984).

A78 ·130/1/-7 See also "Varieties of Self-Reference," included here as chapter 6.

A79 ·131/1/3 This statement remains true: I *still* intend to develop a reflective descriptive system. Cf. the discussion in §6 of the Introduction, and in the Cover.

A80 ·133/1 This paragraph contains glimmers of the intuitions that will form the basis of the proposed fan-calculus, discussed in §6 of the Introduction, and in the Cover.

A81 ·134/0 It was not until 1987 that Rodney Brooks made his famous statement that the "representation" should be discarded in Artificial Intelligence systems—in favour of a view that, in his words, treated "the world as its own best model";[40] see also his "Intelligence Without Reason" and "Intelligence Without Representation."[41] What I take to be significant about the widely-heralded "sea-change" ushered in by the work of Brooks and others[42] is the fact that it betrays what I am here attributing to Weyhrauch: a somehow tacit but deep assumption that "representation" meant constructing within the machine *a replica of the world as a whole* that could be used in its place—as opposed to what cognitive scientists and philosophers of mind take a representational theory of mind to involve, which is that a person "represents" the world only in the sense of employing some interpreted symbols or structures with semantic content involving facts, entities, and states of affairs in the world. Even an internal structure with content along the lines of "Make sure you look out constantly and check the intersection to make sure that it is empty Reification: Reflection Without Metaphysics" would count as a representation on the latter, but apparently not the former, view.

It is hardly surprising that the "full simulation" view of representation needed to be eschewed—though to take that as a rejection of representation altogether is both a rather extreme (and even binaristic) reaction. Brooks later softened his view, saying that systems should use representation "only when necessary"—which opens the door to what representation had originally meant.

For more on Brooks and on the circumstances in which, in my view, representation is required, etc., see "Rehabilitating Representation," ch. 7 of Volume II."

A82 ·135/-1/-5:-1 This dissertation was written in 1981 in Bravo, the first "WYSIWYG"

40. Brooks 1987.

41. Brooks 1991a & 1991b.

42. «Cf. my entry in the MITECS Encyclopedia»

("what you see is what you get") document preparation system, implemented on the Xerox "Alto" minicomputer—arguably the first personal computer ever built, developed in the early 1970s at the Xerox Palo Alto Research Center (PARC). The first 3Lisp implementation was developed in MacLisp, a dialect of Lisp implemented under "ITS" ("Incompatible Time-Sharing System") at the Artificial Intelligence Laboratory at MIT, running on Digital Equipment Corporation PDP-6 and PDP-10.

A83 ·136/0/-1 Sure enough, the implementation listed in the appendix to the dissertation did contain a serious bug. Though it handled reflective procedures correctly, and in general constructed, passed around, and called continuations appropriately, it failed to deal correctly with the rare case of reflective continuations called in the course of normally processed code (which, from a level-shifting point of view, require an instantaneous double level shift). That bug was corrected in the implementation presented in "Implementation of Procedurally Reflective Languages," included here as ch. 5.

Procedural Relection in Programming Languages

3c Lambda Abstraction and Procedural Intension†

Atoms, as I said in [dissertation] section 4a, are used in 2Lisp as context dependent names. I also made clear, both in that section and in [dissertation] chapter 3, that they are taken to designate the referent of the expression to which they are bound. Finally, I have said that they will be statically scoped. A1
It is appropriate to look at all of these issues with a little more care.

The semantical equation governing atoms was given in [dissertation] section 4.a.iii, repeated here: A2

$$\forall e \in \text{ENVS}, f \in \text{FIELDS}, c \in \text{CONTS}, a \in \text{ATOMS} \quad [1] \quad \text{A3}$$
$$[\Sigma(a, e, f, c) = c(e(a), \varphi ef(e(a)), e, f)]$$

If we discharge the use of the abbreviatory φ, this becomes:

$$\forall e \in \text{ENVS}, f \in \text{FIELDS}, c \in \text{CONTS}, a \in \text{ATOMS} \quad [2]$$
$$[\Sigma(a, e, f, c) = c(e(a), \Sigma(e(a), e, f, [\lambda{<}s, d, e_1, f_1{>}\ .\ d]), e, f)]$$

Because all *bindings* are in normal-form, the above equation can be proved equivalent to the following:

$$\forall e \in \text{ENVS}, f \in \text{FIELDS}, c \in \text{CONTS}, a \in \text{ATOMS} \quad [3]$$
$$[\Sigma(a, e, f, c) = \Sigma(e(a), e, f, c)]$$

This is true because, if $e(a)$ is normal, then it will not affect the e and f that are passed to it. Nonetheless, [2] must stand as the definition; [3] as a consequence.

What I did not explain, however, is how environments are constructed. The answer, of course, has first and foremost to

†Section 4·c·i, pp. 377–392, of *Procedural Reflection in Programming Languages*. A link to an internet version of the dissertation is on p. ·.

do with λ-binding. A full account of the significance of atoms and variables, therefore, must rest on the account of the significance of λ-terms. In brief, a λ-term is a complex expression A4 that designates a function. Structurally, in 2Lisp it is any reduction (pair) formed in terms of a designator of the primitive lambda closure and three arguments: a *procedure type*, a *parameter list*, and a *body expression*. The primitive lambda closure is the binding, in the initial environment, of the atom LAMBDA, although there is nothing inviolate about this association. The procedure type argument is typically either EXPR or IMPR (for *extensional procedure* and intensional procedure, respectively; I will discuss these terms more below). The parameter list is a pattern against which arguments are matched, and the body expression is an expression that, typically, contains occurrences of the variables named in the parameter pattern. Thus I am assuming LAMBDA-terms of the following form:

(lambda *procedure-type parameters body*) [4]

I have of course used λ-terms throughout the dissertation, both in Lisp and in the meta-language. It is important, however, not to be misled by this familiarity into thinking we either understand or have yet encountered the full set of issues having to do with λ-abstraction. For this reason the following discussion is framed as if LAMBDA were being introduced for the first time. In this spirit, it is helpful to start by reviewing some simple examples of the use of LAMBDA-terms embedded in larger composite expressions—without any of the complexities of global variables, top-level definitions, recursion or the like. These examples are similar in structure to the kind of term that can be expressed in the λ-calculus (using '⟹', as always, to mean 'normalises to'):

((lambda expr [x] (+ x 1)) 3) ⟹ 4 [5]

((lambda expr [f] [6]
  (f (f 3 4) (f 5 6))
 +) ⟹ 18

```
((lambda expr [g1 g2]                                          [7]
     (g1 (= (nth 1 '[$t])
            (nth 1 ['$t]))
         (g2 [10 20 30])
         (g2 '[10 20 30])))
 if
 (lambda expr [r] (tail 2 r)))     ⟹[30]
```

[5] is a standard example, of the sort 1Lisp would support: the expression (LAMBDA EXPR [X] (+ X 1)) designates the increment function. [6] illustrates the use of a function designator as an argument, making evident the fact that 2Lisp is higher order. Finally, [7] shows that procedurally intensional designators or IMPRs (IF) can be passed as arguments as readily as EXPRs.

There is nothing distinguished or special about these LAMBDA terms, other than the fact that LAMBDA designates a primitive closure. Unlike standard Lisps and the original λ-calculus, in other words, in 2Lisp the label LAMBDA is not treated as a syntactic mark to distinguish one kind of expression from general function applications. Like all pairs, LAMBDA terms are reductions, in which the procedure to which LAMBDA is bound is reduced with a standard set of arguments. I will show below that LAMBDA is initially bound to an intensional procedure, but, as the following example demonstrates, this fact docs not prevent that closure from itself being passed as an argument, or bound to a different atom:

```
(((lambda expr [f]                                             [8]
      (f expr [y] (+ y y)))
  lambda)
 5)                                  ⟹ 10
```

It happens that EXPR also names a function; thus is even possible to have such expressions as:

```
(((lambda expr [funs]                                          [9]
      ((nth 2 funs) (nth 1 funs) [y] (+ y y)))
  [expr lambda])
 5)                                  ⟹ 10
```

Finally, as usual it is the normal-form closures, rather than their names in the standard environment, that are primitively recognised:

```
> (define beta lambda)                                    [10] A5
> beta
> (define standard expr)
> standard
> ((beta standard (f) (f f)) type)
> 'function
```

LAMBDA, in other words, is a **functional**: a function whose range is the set of functions:

```
(type lambda)                    ⟹  'function          [11]
(type (lambda expr [x] (+ x 1)))  ⟹  'function
```

Similarly, EXPR is a function, although I will show how it can be used in function position only later:

```
(type expr)                      ⟹  'function          [12]
```

Though the examples just given illustrate only a fraction of the behaviour of LAMBDA that I will ultimately need to characterise, some of the most important features are clear.

First, LAMBDA is first and foremost a *naming* operator: moreover, the *procedural* import of LAMBDA terms in this or any other Lisp arises not from LAMBDA alone, but from general principles that permeate structures of all sort, and from the type argument I have here made explicit as LAMBDA's first argument. In what follows I will explore the procedural significance of LAMBDA terms at length, but it is important to enter into that discussion fully recognising that it is the body expression that establishes that procedural import, not LAMBDA itself. A6

Second, LAMBDA is itself an *intensional* procedure; neither the parameter pattern nor the body expression is processed when the LAMBDA reduction is itself processed. This is clear in all of the foregoing examples: the parameters—the atoms

that will be bound when the pattern is matched against the arguments, as discussed below—are unbound when the LAMBDA term itself is normalised; but the LAMBDA term does not generate an error when processed. This is because neither the pattern nor the body is treated extensionally—i.e., as being in what is called an "extensional context." (Less clear, although hinted by [9], is the fact that the *PROCEDURE-TYPE* argument to LAMBDA *is* processed at reduction time.)

Further evidence of LAMBDA's procedural intensionality with respect to its second and third argument position is provided in this example:

```
> ((lambda expr [fun]                                    [13]
       (block (print 'last) (fun 1 2)))
   (block (print 'shoe) +)) shoe last
> 3
```

In other words processing of the argument to the LAMBDA term occurred *before* processing of the body internal to that term. The body of a LAMBDA term is then processed each time the function it designates is applied. This fundamental fact about these expressions will motivate the semantical account.

In spite of LAMBDA's intensionality, however, there is nevertheless an important sense in which the context in which the LAMBDA term is itself reduced affects, or at least is relevant to, the behaviour of the resultant procedure when it is used. In A7 particular, we have the following:

```
((lambda expr [fun]                                      [14]
    ((lambda expr [y]
        (fun y))
     2))
 ((lambda expr [y]
     (lambda expr [x] (+ x y)))
  1))                                  ⇒ 3
```

In this example, the atom FUN is bound to a closure designat-

ing a function that adds 1 to its argument. This is because the Y in the body of the lexically last λ-term in the example (the second last line) receives its meaning from the context *in which it was reduced* (a context in which Y is bound to 1), not from the context in which *the function it designates is applied* (a context in which Y is bound to 2). In a dynamically scoped system, [14] would of course reduce to 4.

The expression in [14] is undeniably difficult to read. I will adopt a 2Lisp LET macro, similar to the 1Lisp macro of the same name, to abbreviate the use of embedded LAMBDA terms of this form (this LET will be defined in [dissertation] section 4.d.vii). In particular, expressions of the form

(let [[*param*$_1$ *arg*$_1$] [15]
[*param*$_2$ *arg*$_2$]
…
[*param*$_k$ *arg*$_k$]]
body]

will expand into the corresponding expressions

((lambda expr [*param*$_1$ *param*$_2$ … *param*$_k$] [16]
body)
arg$_1$ *arg*$_2$ … *arg*$_2$]

Similarly, I will define a "sequential LET", called LET*, so that expressions of the form

(let* [[*param*$_1$ *arg*$_1$] [17]
[*param*$_2$ *arg*$_2$]
…
[*param*$_k$ *arg*$_k$]]
body]

will expand into the corresponding expression

```
((lambda expr param1                                          [18]
    ((lambda expr param2
       ...
          ((lambda expr paramk body)
           argk)
       ...)
     arg2))
 arg1]
```

Thus in a use of LET* each arg_i may depend on the bindings of the parameters before it. The difference between these two is illustrated in:

```
(let [[x 1]]                                                  [19]
   (let [[x (+ x 1)]
         [y (- x 1)]]
     y))                          ⟹ 0

(let [(x 1]]                                                  [20]
   (let* [[x (+ x 1)]
          [y (- x 1)]]
     y))                          ⟹ 1
```

Although some of the generality of LAMBDA is lost by using this abbreviation (all LETs and LET*s, for example, are assumed to be EXPRs—i.e., extensional LAMBDAs), I will employ LET and LET* forms widely in subsequent examples. The expression in [14], for example, can be recast using LET, generating an expression much easier to understand, as follows:

```
(let [[fun                                                    [21]
        (let [[y 1]]
          (lambda expr [x] (+ x y)))]]
   (let [[y 2]] (fun y)))                    ⟹ 3
```

The behaviour demonstrated in [14] and again in [21] is of course evidence of what is called *static* or *lexical* scoping; if [14] or [21] reduced to the numeral 4 we would say that *dynamic* or *fluid* scoping was in effect.

The concepts of dynamic and static scoping, however, are by and large described in the literature in terms of *mechanisms* and/or behaviour: one protocol is treated this way; the other that. It is not my policy, in this entire exercise, to accept behavioural accounts as explanations. Throughout, I am committed to being able to answer such questions as "*Why* do these scoping regimens behave the way that they do?" and "*Why* was static scoping used in 2Lisp and 3Lisp?"

Fortunately, the way we have come at these issues leads to a much deeper characterisation of what is going on. In particular, I said that LAMBDA was *intensional*, but example [21] makes it clear that it is not *hyper-intensional*, in the sense of treating its main argument—the body expression—purely as a structural or textual object. It is not the case, in other words, that the reductions involving the function bound to FUN in the third line of [21] consist in the replacing, as a substitute for the word term 'FUN', the textual object '(+ X Y)'. To treat it so would yield an answer of 4—i.e., would imply that 2Lisp has adopted dynamic scoping. Rather, the behaviour demonstrated in [21] shows that what is bound to FUN is neither the body itself, as a textual entity, nor the result of *processing* the body, but rather something intermediate. In ways that we need to understand, what is bound to FUN is an object that in some sense is closer to, or anyway can be associated with, the *intension of the body at the point of the original reduction*.

If we had an adequate theory of intensionality, it might be tempting to say something like the following: that LAMBDA is an [intensional] function from *textual objects* (the body expression and so forth) onto the *intension* of those textual objects in the context in force at the time of reduction. The subsequent use of such a procedure would then "reduce" (or "apply", or whatever intermediate term was chosen as proper to use for combining functions-in-intension with arguments) this intension with the appropriate arguments. There is something right

about this, though for two reasons we cannot let it stand as is. Sadly, first, we have no such theory of functions-in-intension to express it in terms of. Second, it is not quite right, anyway. LAMBDA is of course a function from textual objects *onto functions*, as was made clear earlier; what I need to show, rather, is A8 that (and how) the functions onto which LAMBDA maps its textual arguments somehow preserve, in a context-*independent* way, the potentially context-*dependent* intension of the textual argument in the original context.

Moreover, we can also see that a statically scoped LAMBDA, of the sort constitutive of 2Lisp and 3Lisp, is a *coarser-grained* A9 intensional procedure than is a dynamically scoped LAMBDA. That is:

T1 *Static scoping corresponds to an intensional abstraction operator; dynamic scoping, to a hyper-intensional abstraction operator.*

In order to understand T1 in depth, we need to retreat a little from the rather behavioural view of LAMBDA that I have been presenting, and look more closely at what λ-abstraction consists in from the original perspective of its being a naming operator. It is all very well to show how LAMBDA terms *behave*, in other words; but we have not yet adequately answered the question "What do LAMBDA terms *mean*?"

Speaking extensionally, LAMBDA terms designate functions; that much is clear. We also know that functions are sets of ordered pairs, such that no two pairs coincide in their first ele- A10 ment. We know, too, what application is: a function applied to an argument is the second clement of that ordered pair in the set whose first element is the argument. However none of this elementary knowledge suggests any relationship between a function and a function *designator*. And until we understand that relation, we will not be in a position to understand, intensionally, that designator's *meaning*.

Informally, we have a consensual intuition about λ—that it is an operator over a list of variables and expressions, designating the function that is signified by the λ-abstraction of the given variables in the expression that is its "body" argument. However this intuition—including its telling use of the phrase 'λ-abstraction'—must arise independently of any of the extensional points made in the preceding paragraph. To understand the meaning of a λ-term, therefore, requires an analysis of it *as a term*.

The fundamental intuition underlying λ-terms and λ-abstraction in general can be traced at least as far back as Frege's study of predicates and sentences in natural language. In particular, I believe that it is best to understand a λ-term is as a *designator with a hole in it*, just as Frege understood a predicate term as a *sentence with a hole in it*. If, for example, we take (and assume to be true) the sentence "*Mordecai was Esther's cousin*," and delete the first designating term, then we obtain the expression "_____ *was Esther's cousin*." It is easy to imagine constructing an infinite set of other derivative sentences from this fragment, by filling in the blank with a variety of other designating terms. Thus for example we might construct "*Aaron was Esther's cousin*" and "*the person who lives across the fjord was Esther's cousin*" and so forth. In general, some of these constructed sentences will be true, and some will be false. In the simplest case, also, the truth or falsity hinges not on the actual form of the designator we insert into the blank (whether we say 'the person who lives across the fjord' or 'the person who was here for tea yesterday'), but on the *referent* of that designator. Thus our example sentence will be true if the supplied designator refers to Mordecai; any term codesignative with the proper name "Mordecai" would serve equally well.

Predicates arise naturally from consideration of *sentences* containing blanks; that was Frege's insight. The situation regarding *designators* containing blanks—and the resultant functions—is entirely parallel. Thus if we take a complex noun

phrase such as "*the country immediately to the south of Ethiopia*," and remove the final constituent noun phrase, we get the open phrase "the country immediately to the south of _____." Once again, by filling in the blank with any of an infinite set of possible terms (designating noun phrases), the resultant composite noun phrase will (perhaps) designate another object. In those cases where the resultant phrase succeeds in picking out a unique referent, we say: (i) that the object so selected is in the *range* of what is designated by the phrase that contained the blank; and (ii) that the object designated by the phrase we insert into the blank is in that entity's *domain*. In this way we erect the entire notion of *function* with which we are so familiar. A11

Once this basic approach is adopted, a raft of more specific questions arise. What happens, for example, if we construct a phrase with *two* blanks? The answer, of course, is that we are led to a function of more than one argument. What if the noun phrase we wish to delete occurs more than once (as for example the term 'Ichabod' in "The first person to like Ichabod and Ichabod's horse")? The power of the λ-calculus can be seen as a formal device to answer all of these various questions. In particular, we can understand the formal *parameters* as a *method of labeling the holes*: if one parameter occurs in more than one position within the body of the lambda expression, then tokens of the formal parameters stand in place of a single designator that had more than one occurrence. If there is more than one formal parameter, then more than a single noun phrase position has been made "blank." And so on and so forth—all of this is familiar.

It is instructive to review this history, for it leads to a particular stance on some otherwise difficult questions. Note for one thing how it clarifies a point we started with: that the function of LAMBDA as a first and foremost a *naming* operator. In addition, it is important to recognise how *syntactic* a characterisa-

tion this has been: I have talked almost completely about signs and expressions (terms, phrases, etc.), even though we realised that the semantical import of the resultant sentences or completed noun phrases depended (in the simple extensional case) only on the referents of the noun phrases that were inserted into the blank(s). It was Frege's technique to motivate the abstract ontological notions of *predicates*, *relations*, *functions*, etc. as derivative on such syntactic manoeuvring. The technique is important in the present case because it gives us a stance from A12 which to ask essentially syntactic or structural questions in order to get at the ontological intuitions behind λ-abstraction (indeed, it is because I want the structural answers to these questions that I am pursuing this whole line of thought).

Suppose, then, to stay with the case of defining predicates, that we wish to define (i.e., name) a predicate by inserting a blank into some otherwise complete sentence—i.e., by deleting a noun phrase from it. What context, we may ask, determines the meaning of the resulting open expression? The only plausible answer that honours the referential character of naming is *the context in which the definition was originally introduced*. Suppose, for example, that while writing this paragraph I utter the sentence "*Bob is going to vote for the President's eldest daughter*." Again staying with the simplest case, it is natural to assume that I refer to the (current) President's eldest daughter, known by the name "Maureen Reagan." If I excise the noun "Bob" and construct the open sentence "_____ *is going to vote for the President's eldest daughter*," then I have constructed a predicate true of people who will vote for Maureen Reagan. That is, the interpretation of "the President's eldest daughter" is determined by the context where the predicate was introduced. This, at least, is the simplest and most straightforward reading. It would undeniably be more complex, even if one A13 could nonetheless argue that it would be logically coherent, to suggest that what is designated hyper-intensionally involves

the whole open sentence *qua sentence*—so that when we asked whether the resultant predicate is true of some person we would determine the referent of the phrase *"the President"* only at that point. The ground intuition is unarguably extensional.

What does this suggest regarding Lisp? Simply this—that A14 A15
the natural way to view λ-terms is as:

1. Expressions that designate functions, derived from
2. Composite referring terms, in which one or more ingredient terms have been replaced by blanks, where
3. The *parameters* are a formal mechanism to label the blanks, so as to facilitate a subsequent process of filling the blanks in with other terms, and
4. Where the function designated is determined *with respect to the context of use where the LAMBDA term is stated or introduced* (as opposed to where the designated function is subsequently applied).

Notably, the foregoing four points *lead us* to an adoption of statically scoped free variables, because we can show how that procedural mechanism correctly captures the original (declarative) naming intuition. In other words, I am claiming that: A16

T2 *Static scoping is the truest formal reconstruction of the (ultimately referential) linguistic intuitions and practice upon which the notion of λ-abstraction is based.*

In general, in order to remain true to Church's λ-calculus, we must be true to the understanding that his calculus embodies, rather than slavishly mimic its operations. This mandate has further-reaching consequences than those articulated in T2. In particular, to propose a full *substitutional* procedural regimen for 2Lisp would be crazy—it would be to *mimic his mechanism*, rather than *accomplish what his mechanism was for*. Since 2Lisp is a formalism with procedural side-effects, such a regime would imply that every occurrence of a formal parameter

within a procedure body would engender another instance of any side-effects implied by an argument expression. This was not a problem for Church because the λ-calculus of course has no side-effects. A17

In sum, I will insist that the term

```
(let [[y 1]]                                    [22]
   (lambda expr [x] (+ x y)))
```

designate the increment function, rather than designating that function that adds to its argument the referent of the sign "y" in the context of use of the designating procedure.

As far as it goes, this is straightfoward. I showed in [dissertation] chapter 2 how the static reading leads naturally to a higher-order dialect, to uniform processing of the expression in "function position" in a redex, and so forth, though in that chapter I did not do what we have done here: examine the underlying semantical motivation for this particular choice. Nor, in that context, did I explicitly examine another subject to which we must now turn: *the intensional significance of a* LAMBDA *term*. That this further question remains open is seen when one realises that the preceding discussion argues only that the *extension* of the LAMBDA term be determined by the context of use in force at the point where the LAMBDA term it introduced. However I have not yet examined the full computational significance of the term in "body" position—i.e., to use the reconstruction I am recommending, the full computational significance of the open designator containing demarcated blanks.

For pointers, it is again instructive to look at the reduction regimen that Church adopted for the λ-calculus. As I have said, the λ-calculus is a statically-scoped higher order formalism. By the discussion just advanced, the λ-calculus should depend on an intensional LAMBDA, but of course no theory of

"functions in-intension" accompanies theoretical treatments of the λ-calculus. This is related to the fact that, in the λ-calculus, the item "λ" is not itself considered to be in a function-designating position. This is because the λ-calculus is strictly an *extensional* system; there is no way in which an appropriately *intensional* function could be defined within its boundaries. It is thus effectively a *necessary* rather than contingent fact that λ-terms in the λ-calculus are demarcated *notationally*, as they were in the first version of 1Lisp that I presented in [dissertation] chapter 2. (In the λ-calculus, that is, the "λ" is a syncategorematic uninterpreted mark, on a par with parentheses and dots). A18

In order to understand the λ-calculus and λ-abstraction more generally, it is essential to recognize that its substitutional reduction regime is defined within this set of constraints. Superficially, after all, substitution is a hyper-intensional kind of practice. During β-reductions, *actual textual expressions* are substituted, one within another, during the reduction of a composite λ-calculus term. This would appear to conflict with the claim made above in T1: that hyper-intensional abstraction corresponds to dynamic scoping, and intensional abstraction to static or lexical. How then can I defend my claim of intensional abstraction in a statically scoped formalism, and yet use the λ-calculus as a motivating example?

The answer is that the λ-calculus has been crafted in such a way as to enable its hyper-intensional substitution protocols A19 to *mimic* or *implement* the more abstract intensional behaviour that ideally, if we had a adequate theory of functions-in-intension, we would be able to define more directly. In particular, three properties of the λ-calculus contribute to this ability. First, as already mentioned: the λ-calculus is *extensional*, the mark 'λ' is not used as a term (i.e., not in function position), and no facility is provided for the user to construct intensional functions. Second, there is no primitive quotation operator in

the λ-calculus (and of course no corresponding mechanism of disquotation), so that it is not possible in general and unpredictable ways to capture an expression from one context and to slip it into the course of the reduction in some other place ("behind the back of the reduction rules," so to speak)—a practice that genuinely would engender something like dynamic scoping. Third, as well as superficially involving hyper-intensional β-reduction, the λ-calculus also depends on a seemingly pesky but in fact critically important additional rule, having to do with variable capture, called α-reduction. It is a constraint on β-reduction—the main reductive rule in the λ-calculus—that terms may not be substituted into positions *in such a way that any open (unbound) variables would be "captured" by an encompassing* λ*-abstraction*. If such a capture would arise, one is obligated first, using α-reduction, to rename the parameters involved in such a fashion that the capture is avoided. The following, for example, is an incorrect series of β-reductions:

```
(λf . ((λg . (λf . fg)) f))      ; an illegal derivation      [23]
(λf . (λf . ff))
```

Rather, one must use an instance of α-reduction to rename the inner f so that the substitution, for g, of the binding of g will not inadvertently lead that substitution to "become" an instance of the inner binding. Thus the following is correct:

```
(λf . ((λg . (λf . fg)) f))      ; a legal derivation         [24]
(λf . ((λg . (λh . hg)) f))      ; first an α-reduction
(λf . (λh . hf))                 ; then a valid β-reduction.
```

In other words α-reduction is expressly designed, from a procedural point of view, to ensure that, in those cases where the context of *use* of a λ-term might conflict with the context of *introduction*, the λ-term is adjusted so that the function it designates remains uninfluenced. That is: the role of α-reduction in the λ-calculus is to rearrange textual objects so as to *avoid*

the dynamic scoping that would be implied if α-reduction did not exist.

Together, in sum, these three conditions ensure that, in spite of β-reduction's hyper-intensional character, the λ-calculus' overall procedural regimen honours the conditions of static or lexical scoping.

We are still not done. We need to ask *why* the reduction in [23] is ruled out—*why* dynamic scoping is so carefully avoided. The answer cannot be that the resulting system is incoherent, since, modulo issues of side effects, β-reductions with no α-reductions is one way to view Lisp 1.5 and all its descendants. Sure enough the Church-Rosser theorem would not hold, but, as our experience with these Lisps has shown, one can simply discard that theorem and decide rather arbitrarily on one reduction order. But we now have an answer: dynamic scoping violates the condition we adopted above: that the "meaning" or intension of a λ-term be determined by the context in force in the place where that term is introduced. Variable capture is bad because it *alters* that intension—thereby violating intention.

Thus we have reached the following important conclusion:

T3 *The reduction of LAMBDA terms must preserve, in a context-independent way, the (potentially) context-dependent intension of the body expression.*

This of course is a much stronger result than the overarching mandate that in every case ψ preserve *designation*. In general, reduction (ψ) of composite terms does *not* preserve intension, according to a commonsense notion of intension. This is difficult to say formally, for two reasons. The most serious is the standard one: that we do not have a theory of intension with respect to which to formulate it. If one takes the intension of an expression to be the function from possible worlds onto exten-

sions of that expression in each possible world—the approach taken in possible world semantics and by such theorists as Montague[1]—then it emerges (if one believes that arithmetic is not contingent) that all designators of the same number are intensionally as well as extensionally equivalent. Thus (+ 1 1) and (SQRT 4) would be considered intensionally equivalent to 2 (providing of course we are in a context in which SQRT designates the square-root function). I would argue, however, that this conclusion violates lay intuition—that a more adequate treatment of (even mathematical) intensionality should be finer grained (perhaps of a sort suggested by Lewis[2]). Second, without specifying the intensions of the primitive nominals in a Lisp system, it is difficult to know whether intension is preserved in a reduction. Suppose, for example, that the atom PLANETS designates the sun's planets, and is bound to the rail [MERCURY VENUS EARTH … PLUTO]. Then (CARDINALITY PLANETS) might reduce to the numeral 9 if CARDINALITY was procedurally defined in terms of LENGTH. It is argued, however, that the phrases "*the number of planets*" and "*nine*" are intensionally distinct because "*the number of planets*" might have designated some other number, if there were a different number of planets, whereas, in this language, "*nine*" necessarily designates the number nine. On such an account the reduction of (CARDINALITY PLANETS) to 9 is not intension preserving. A20 A21

Making precise our intuitions about the nature of intensionality in general is not my present subject matter, however. Furthermore, and fortunately, if all we ask of the reduction of LAMBDA terms to normal form is that intension be *preserved*, we do not have to *reify* intensions at all—we do not even have to take a position on whether intensions are *things*. All that we are bound to ensure is the substance of T2: that the intensional character of the expression over which the LAMBDA term

1. Cf. Montague (1970, 1973). [Note: this footnote was numbered 4 in the original version of the disseratation.]
2. Lewis (1972).

abstracts must be preserved, in a context-independent way, in the normal-form function designator to which the LAMBDA term reduces.

At the declarative level this suffices—it will be my guiding mandate in defining the procedural treatment of LAMBDA terms. A further set of questions needs to be answered, however, having to do with the relationship between the intensional content of a Lisp expression and its full computational significance, including its procedural consequence. The issue is best introduced with an example that I will make use of later. It is a widely appreciated fact that, if an expression X should not be processed at a given time, but should be processed at another time, it is standard computational technique to wrap it in a procedure definition, and then to *reduce* it subsequently, rather than simply *using* it. A simple example is illustrated in the following two cases: in the first the (print 'there) happens *before* the call to (print 'in); in the second it happens *after*:

```
> (let [[x (print 'there)]]                                    [25]
      (block (print 'in) x)) there in
> $T
> (let [[x (lambda expr [] (print 'there))]]                   [26]
      (block (print 'in) (x))) in there
> $T
```

Because of 2Lisp's static scoping, which corresponds to this intensional reading of LAMBDA, this approach can be used even if variables are involved:

```
> (let* [[x 'there]                                            [27]
         [y (print x)]]
      (block (print 'in) y)) there in
> $T
> (let* [[x 'there]                                            [28]
         [y (lambda expr [] (print x))]]
      (block (print 'in) (y))) in there
> $T
```

What this example illustrates is that the side-effects engendered by a term (input/output behaviour is the form of side-effect illustrated here, but of course control and field effects are similar) take place *only when the term is processed in an extensional position*. In other words if the reduction of a LAMBDA-term takes (and preserves) an *intensional* reading of the body expression, it does not thereby engender the full computational significance of that expression. Such significance arises only when some other function or context requires an *extensional* reading. Side-effects, that is, can be considered to be parts of the **procedural extension** of a 2Lisp expression.

The QUOTE function in 2Lisp that I defined in 84-132, and handles in general, are *hyper-intensional* operators; it was clear in their situation that the significance of the mentioned term was not engendered by the reduction of the hyper-intensional operator over the term. I have not, however, previously been forced to ask the question of what happens with respect to *intensional* operators, but the examples just adduced yield an answer: their processing, too, does not release the potential significance of the term. Or to put it another way: A22

T4 *The full computational significance of both hyper-intensional and intensional computational expressions does not release the full computational significance latent in their ingredients.*

It is for this reason that the "deferring" technique alluded to above works in the way that it does. (Note, again, that no suggestion is afforded by the λ-calculus with respect to this concern, since that calculus contains no side-effects at all.) Thus we might say that 'intensional' and 'hyper-intensional' are defined not just declaratively, but more generally in terms of full computational significance.

In sum, we have reach the following constraint: *intension-preserving* term transformations do not engender the proce-

dural consequences latent in an expression; those consequences emerge only during the normalisation of an extensional redex, in which case *intension is not (in general) preserved*. Recall that although (+ 2 3) reduces to *co-extensional* 6, it is on my view *not* the case that (+ 2 3) and 6 are intensionally equivalent.

One more question needs to be examined, before I am ready to characterise the full significance of LAMBDA. As noted above, A23
in spite of my claim that LAMBDA is an intensional operator, it cannot be the case that LAMBDA is a function from expressions onto intensions, nor is it the case that LAMBDA terms reduce to intensions. If X is a term (LAMBDA …), in other words, neither φ(X) nor ψ(X) is an intension; both possibilities are rejected by protocols long since accepted. In particular, note that in any 2Lisp form (*F* . *A*), the significance of the whole arises from the application of the function designated by *F* to the arguments signified by *A*. Thus in (+ 2 3), which is in reality (+ . [2 3]), I said that the whole designated five because the atom "+" designated the extensionalised addition function, which when applied to a syntactic designator of a sequence of two numbers, yielded their sum.

Similarly, in any expression

((lambda *type params body*) . *args*) [29]

it follows that the term (LAMBDA …) must designate a function. Similarly, in a construct like

```
(let [[f (lambda ... )]]                    [30]
   (f . args)]
```

F must also designate a function. This is all consistent with the requirement that variable binding be designation-preserving. In [30], F and (LAMBDA …) must be co-designative.

It follows, then, that F cannot *designate* the intension of the (LAMBDA …) term. Hence (LAMBDA …) cannot normalise to

a designator of that function's intension. For we do not know what intensions are, but they are presumably not syntactic, structural entities. They are not, in other words, elements of the set of structural field elements S, and ψ has S as its range. I said earlier, however, that F must be intensionally similar to the LAMBDA term—what this brings out is that F must be **co-intensional** with the LAMBDA term, as well as *co-extensional*. The normalisation of LAMBDA redexes, in other words, must *preserve intension as well as extension*. That is, to put it all together:

T5 *Structures of the form* (lambda ***pattern body***) *are context-dependent function designators. The normalisation of such forms must yield structures that preserve, in a context-independent way, the* full computational significance *of those designators—both intensional and extensional, both declarative and procedural.*

* * *

This is as much as I will say regarding LAMBDA in its simple uses. As usual, in accord with my general approach, I have attempted to characterise LAMBDA terms primarily in terms of *what they mean* (declaratively, as names); from this I have attempted to justify an account of *how they are to behave*. As always, that is, ψ is subservient to φ.

Finally, in terms of this analysis of LAMBDA, I need to say how reduction *works*. The answer is of course quickly stated, and familiar. When a LAMBDA term is reduced, a closure is constructed and returned as the result. When a pair whose CAR normalises to a non-primitive closure is encountered, the closure is *reduced* with the arguments. If that closure is an EXPR, then this reduction begins with the reduction of the CDR of the pair, followed by a process of binding the variables in the parameter pattern to the resultant normal-form argument

designator. If the closure is an IMPR, no argument normalisation is performed; instead a handle designating the CDR of the pair is matched against the parameter pattern. In either case the body of the closure (the body of the original reduction with LAMBDA) is processed in a context that, as usual, consists of a field and an environment The field is the field that results from the processing of the arguments—as usual there is no structure to the use of fields: a single field is merely passed along throughout the computation. The environment, however—this is the mechanism that allows the intension to be that of the point of introduction—is the environment that was in force at the point when the closure was constructed, but augmented to include the bindings generated by the pattern match of arguments against variables.

If we were equipped with a theory of functions in intension, and could therefore avail ourselves of an intensional operator in the meta-language, called INTENSION-OF, that mapped terms and lists of formal parameters into *intensions*—whatever they might be—then we could specify this entire desired semantical import of LAMBDA in its terms. Lacking such a theory, I will instead look at LAMBDA from the point of view of designation and reduction, armed with the mandate that it is the intensional properties of the resultant structures that are of primary concern.

Annotations[1]

A1 ·167/1/3:5 It is true that atoms designate the reference of the expression to which they are bound, but the normative relation of φ and ψ would have been more clearly expressed if the sentence had been written: "they are bound to normal-form designators of their referents."

A2 ·167/2 As stated in the Introduction to the dissertation (ch. 3b), Σ is a *general significance function* that specifies both the co-constituted declarative import (φ) and procedural consequence (ψ) of computational structures. If S is the set (or type) of internal structural elements,[2] ENVS of environments, FIELDS of structural field states,[3] and CONTS of continuations, then, as defined in dissertations chapters 2 and 3:

$$\varphi : [[\text{ENVS} \times \text{FIELDS}] \rightarrow [S \rightarrow D]]$$
$$\psi : [[\text{ENVS} \times \text{FIELDS}] \rightarrow [S \rightarrow S]]$$
$$\text{CONTS} : [[S \times \text{ENVS} \times \text{FIELDS}] \rightarrow [S \times \text{ENVS} \times \text{FIELDS}]]$$
$$\Sigma : [[S \times \text{ENVS} \times \text{FIELDS} \times \text{CONTS}] \rightarrow [S \times \text{ENVS} \times \text{FIELDS}]]$$

A3 ·167/3/1 In the original dissertation the equations in this section were numbered s4-430 through s4-459. For this version I have renumbered them 1-30.

A4 ·168/0/3 By the phrases "λ-term" and "LAMBDA-term," throughout this section, I do not mean the singleton term 'λ' or 'LAMBDA', but rather what it would have been better to call a "λ-redex" (or "LAMBDA-redex)—i.e., *composite* function-designating term of the form '(λ*v* . *expression*)' in the λ-calculus or '(LAMBDA *TYPE PATTERN BODY*)' in 2Lisp.

In general, I use the version "λ-term" both for such redexes specifically in the λ-calculus or when I wish to refer to instances of both kinds, and "LAMBDA-term" when making specific reference to these structures in 2Lisp.

A5 ·170/2/1 As had been explained in an earlier dissertation chapter, in transcribing interactive 2Lisp sessions, I used '>' as a prompt (before both input and output), and italics to signal user input.

A6 ·170/-2 This paragaraph is somewhat disingenuous. It is of course true that the procedural consequence of LAMBDA-terms does not arise from the *name* 'LAMBDA', though it is definitely associated with the primi-

1. References are in the form *page/paragraph/line; with ranges (of any type) indicated as x:y. For details see the explanation on p.·*

2. What in «where?» I call *impressions*.

3. I.e., a determination of all facts pertaining to issues that are amenable to side-effects, such as the elements of all pairs and rails.

tive LAMBDA closure. What I believe I meant, though, is that the specific procedural consequence of normalising a LAMBDA term (i.e., LAMBDA redex) has to do with facts about the body. Specifically, as I explain later in the text, the procedural mandate on LAMBDA redexes is that, when normalised, they need to *preserve the intension* of the body expression.

A7 :171/4/-2 As explained more clearly in the paragraph following example [14], in saying "the behaviour of the resultant procedure *when it is used*" (emphasis added), I mean when the procedure defined or named by the lambda term is subsequently, as it is said, "applied."

A8 :175/0/4:5 In saying that LAMBDA is "a function from textual objects onto functions" I mean that the range of LAMBDA is the set of functions whose intensions we are currently discussing; the range is not those intensions themselves.[4]

A9 :175/1 That 2Lisp intensions are coarser-grained than (hyper-intensional) 2Lisp text stems in part from the fact that 2Lisp does not deal in any very serious way with issues of deixis and indexicality (cf. 03).

Thesis T1 is ultimately less important than T5, on p 188.[5] It is the intension-preservation, not the coarse-grainedness, that is important about statically-scoped LAMBDAS.

A10 :175/-1/2:3 I no longer believe that functions *are* sets of ordered pairs; merely that we standardly *model* them as sets of ordered pairs. Cf. "The Correspondence Continuum" (ch. 11)

A11 :177/0/-7 'Perhaps' because there may be no country south of what is designated by the term that fills in the blank—if, for example, it were 'Chile.'

A12 :178/0/7:9 The preternatural intimacy of connection between the compositional structure of language (noun phrases, verb phrases, etc.) and the standard ontological structures of the world (objects, properties, relations, etc.) had haunted me since the 1960s. Cf. the discussion in 03 (p. 284) of whether God made the compositional structure of *world* on Tuesday, and of *word* on Friday—or whether, as I ultimately claim in that book, they are a two aspects of a single fact, to be accounted for in an integrated notion of representation-*cum*-ontology.

4. Someone mathematically inclined might suggest modeling intensions *as* functions of some sort; the point would be that the range of LAMBDA is not those functions-used-to-model-intensions, but rather the functions whose intensions those functions are being used to model.

5. The labels 'T1'–'T5' are not in the original dissertation; they were introduced for this version, to facilitate cross-reference.

A13 ·178//-3 As I knew perfectly well at the time, the example shows that *temporal*, *locational, discursive*, and other forms of context matter as much as issues of *lexical position*. In 2012, if uttered in the U.S., "the President's eldest daughter" would refer to Malia Obama; if used in France, to Giulia Sarkozy, etc.

A14 ·179/0 As linguists and philosophers of language know only too well, issues of the appropriate understanding of intensionality in natural language are vastly more complex than this paragraph suggests. It was not until later in the 1980s that I learned enough linguistics to understand how amateur this paragraph really is.

Not only are the issues complex; there are deep underlying issues of what gets decided when, if ever. Imagine, for example, that in 2012 I ask someone whether, if they were President of the U.S., they would expect the First Lady to attend gala events. Even if the question seems fully understood, it can remain unclear whether the constituent singular term 'the First Lady' was intended to refer to Michelle Obama, to their current female partner, or to the person who would be their female partner if that fictitious scenario were to come to pass. What is perhaps most striking about the example is how seemingly intelligible the question is without this issue being resolved, *even by the questioner*—suggesting that the "meaning" or "intension" of the question, if reifying it even makes sense, may not even require that such issues be settled. Or to take a different kind of example, consider the sentence "After the team's ignominious defeat, every fan in the city took refuge in a local bar,"[6] where the interpretation of the term 'local' is not determined by the locale relevant to the uttering of the sentence, but is (at least presumably) quantified so as to be depend on each separate fan. Does that mean that in this sentence 'local' occurs in an intensional context? And so on.

Intensional issues in computation are equally complex. As perhaps the simplest non-trivial example, consider so-called "left-hand values"—as for example in code fragment "x[3]=7", intended to set the third element of array x to 7. Neither the term 'x[3]' in the computational expression—nor, for that matter, the term "the third element of array x" in the English gloss—is straightforwardly in an "extensional position," in the sense of referring to what the same computational term would refer to if used in a "right-hand side"

6. A sentence in famous among linguists; «ref».

context, such as "27+x[3]," or the same English phrase would refer to in a statement "the third element of the array is 49."

A15 ·179/1/1ff This sentence has been expanded from the original in the dissertation to enumerate these four points explicitly, so to make its original meaning clear.

A16 ·179/-3/2:4 Cf. §5c of the Introduction.

A17 ·180/0/0:2 For example, under a substitutional regimen the expression:

```
((LAMBDA EXPR [F] (+ F F F)) (PROGN (PRINT "hello") 3))
```

would print "hello" three times before returning 9.

A18 ·181/0/-3:-1 Cf. note 9 in annotation A14 of ch. 3b, on p. ·142.

A19 ·181/-1 This paragraph has been more than usually edited, for this version, in order to make its points clearer.

A20 ·184/0/-8:-7 In conjunction, presumably, with a commitment to a closed-world assumption. Why the example was framed so procedurally, however, I have no idea.

A21 ·184/0/-4:-3 Notwithstanding this claim, I, like many others, would have been astonished, in 1981, if someone had foretold that in 2012 the number of planets would have been reduced to 8.

A22 ·186/1/1 The label 's4-132' refers to a definition on p. 288 of the full dissertation; for a link to an internet accessible version cf. p. ·.

A22 ·187/1/2 Re 'as noted above': cf. 175/0/3:9.