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 behavior. 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 a reflective dialect of Fortran, Smalltalk, or any other procedural calculus, by pursuing essentially the same approach as I will demonstrate here for Lisp. A5

The first Lisp dialect (called **1Lisp**) will be an example intended to summarize 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 behavior of* (what computer science calls) *the "interpreter."* The interpreter is then subsequently defined with respect to this theory of attributed semantics, so that the result of processing of an expression—i.e., the value of the function computed by the basic interpretation pro- A6

cess—is a *normal-form co-designator* of the input expression. I will call 2Lisp a **semantically rationalized** 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 A9
I will claim characterizes 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 A10
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 A11
in conjunction).

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 declar- A12
ative 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

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 1Lisp "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 rationalized** 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 rationalized* 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 analyzed: (i) notation; (ii) abstract structure; (iii) 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. A13

Category alignment is an aesthetic with consequence. I will show that the 1Lisp programmer (i.e., all existing Lisp programmers) must in certain situations resort to meta-syntactic machinery merely because 1Lisp fails to satisfy this mild requirement (in particular, 1Lisp *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 rationalized 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 characterization 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 behavior, 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 behavior 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 ω-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 behaviors 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 rationalized* 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 rationalized 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 behavior, 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 behavior 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 behavior—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 naturalizing 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 behavior 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 behavior (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. A30

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 behavior. 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 [1] to constrain its behavior, 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 [1]. 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 behavior 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 realize 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 behaviors 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 behavior is our ultimate subject matter, not simply the mathematical characterization of formal relationships. A31

In dealing with computational processes, we are dealing with artefacts *behaviorally* 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 analyze my canoeing skills and develop an account of how I would do better to lean downstream when exiting an eddy. Coming to this realization 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 behavior*—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 behavior, so does *prior* non-reflective behavior 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 capsized.

In sum, the relationship between reflective and non-reflective behavior 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 behavior so that the right causal powers can be transferred across the boundary, without falling into the opposite difficulty of making them so inter-

connected 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. A32

*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 behavior, 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 behavior, even though those entities cannot be claimed to have a theory-independent ontological existence in the behavior 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- A33

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 A34 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 behavior 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 analyzed, 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 behavior; and at the same time (ii) the ability for the user (by using reflection) to be able to modify or adjust the behavior 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] A35

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.

A36

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 behavior. There has been in the last century no lack of investigation into self-ref-

erential 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 A37 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. A38

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:

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

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:

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

The requirement that reflection *matter,* to summarize, 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- A39

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 behavior, 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 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 characterization 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. A40

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 behavior 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 behavior (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' behavior, including the exact temporal relationships between one event and the next, and so forth.

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 behavioral 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 A41 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 behavioral 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 com- A42
putation. 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 realized. I will view these alternatives forms of reduction—from process to, say, behaviors 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 behavior 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 **realization** of a computational process, an entire hierarchy of processes above it may obtain indirect realization 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 realized.

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 behavior is (i.e., can be "interpreted" as) the behavior 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]

A43 A44 A45

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. A46

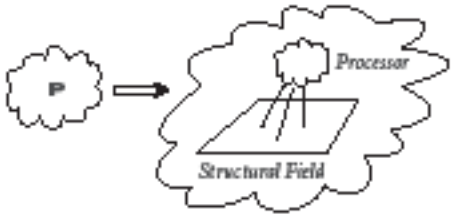

Figure 3

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, A47 we can characterize 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-

. . .

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 this 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 behavior. 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.

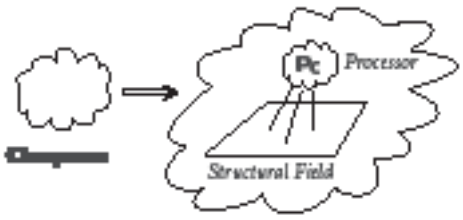

Figure 4

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.

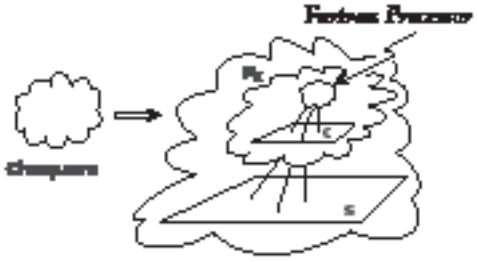

Figure 5

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 behavior 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 behavior 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 realized. 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 behavior—i.e., desired process* Q*.*

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

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

In terms of this definition, we can characterize 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 *behaviorally* 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. A48

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 behavior.)

Finally, it is straightforward to imagine how this model could be used in cognitive theorizing. A **weak** computational model of some mental phenomenon or behavior ψ would be A49 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).

A50

• • •

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. A51

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*. A52

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 behavior 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. A53

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 A54 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 predi- A55 cate on explanations, not on artefacts.

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 "syntacti- A56 cally"—that we are liable to forget that it is a major claim, and are in danger of thinking that the simpler phrase "symbol ma-

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 for 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 recognized in computer science.

1d The Rationalization 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 behavior 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, A57

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 characterization 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 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 A58

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.* A59

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 tradi- A60
tional formal systems (logic, the λ-calculus, and so forth), but that has been lost in the attempt to characterize 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 A61
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 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 recognized 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 characterize 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 characterization 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. A62

ψ indicates the formally computed relationship—a distinction similar, as I will soon argue, to that between the logical relationships of *derivability* (⊢) and *entailment* (⊨). A63

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. A64

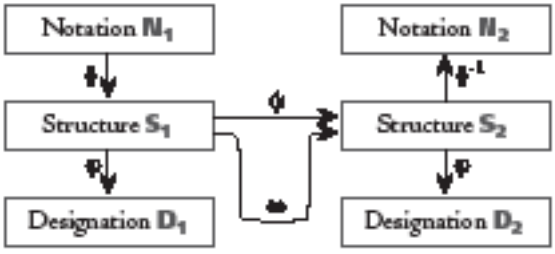

Figure 7

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 A65

13. Fodor (1980).

"the pseudonym of Samuel Clemens," I reply "Mark Twain", A66
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 A67
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 A68
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 characterization 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 A69
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; ⊡ is the set of structural field elements.)

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

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

structures 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).

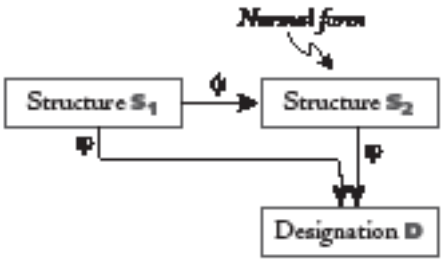

Figure 8

The behavior described in [5] is unfortunate, in part because the question of whether $\varphi(s) \in$? 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):

$$?s \in ? \; [[\phi(\psi(s))=\phi(s)] \wedge [\text{NORMAL-FORM}(\psi(s))]] \qquad [6]$$

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, I will define normal-formedness 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 ψ(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 (ψ=ψ∘ψ—i.e., ∀s ψ(s)=ψ(ψ(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 ψ(s) will be predictable, independent of the form of s itself (although the *token identity* of ψ(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 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

A70

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 recognize that the behavior 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 rationalization 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. A71

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 ef-

fects, 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 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. A72

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 summarize 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 recognized 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 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 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.

A73 A74

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 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:

A75

(+ 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 consequence, and claim that [7] designates seven, even though it pa-

tently returns the numeral 17 (although I am under no obligation to make the declarative and procedural stories cohere—in fact I will reject 1Lisp 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 characterizing the full semantics of standard Lisp (as exemplified in my 1Lisp 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* 1Lisp, not merely to *characterize* 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

A76 A77

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 analyzing 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 characterizations 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 characterized 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 behavior (since they are *programs,* not

processors, strictly speaking); and (ii) the behavior 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 A78 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 behaviorally equivalent to that of the primitive processor itself. In other words, if we were mathematically to take processes as functions from structure onto behavior, and if we name the processor presented in figure 12 MCP_{2L}, and the primitive 2Lisp processor P_{2L}, then if we taken '⍰' to mean behaviorally 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]):

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

It should be recognized 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 behavior 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 behavior 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.

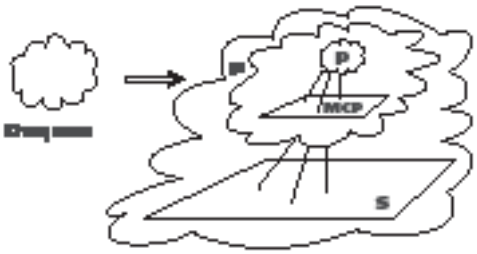

Figure 13

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.

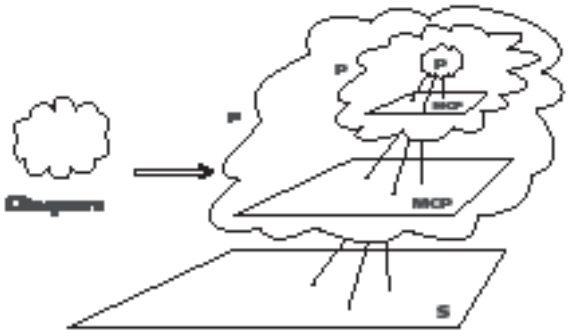

Figure 14

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 processors 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. A79

```
1  (define READ-NORMALISE-PRINT
2  ... (lambda simple [level env stream]
3  .....(normalise (prompt&read level stream) env
4  ..... (lambda simple [result]                                   ; C-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!]                                   ; C-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 characterized 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 reflective 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 purposes A80 one is simpler, for others the other. What matters most is to recognize their behavioral 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 behavior 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 behavior 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 behavior. 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. A81

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 rationalized 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 rationalized 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 procedural calculus—with at least the following properties: A82

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 behavior 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- A83
trivially, Lisp is the *lingua franca* of the AI community; this fact alone makes it an eminent candidate.

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 behavior—such as CATCH and THROW and QUIT—have been added to the repertoire of behaviors 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

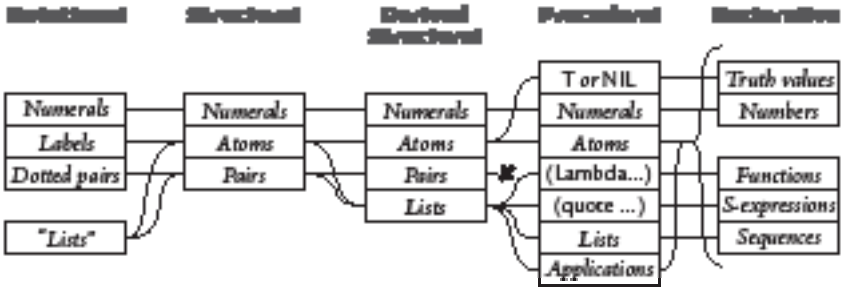

Figure 16 — The Category Structure of Lisp 1.5

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 1Lisp 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 1Lisp'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 1Lisp 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 behavior 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 rationalization, 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 rationalization 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-preserving). Details are provided in [dissertation] chapter 4, but a sense of the resulting architecture can be given here.

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). A84

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 nu-

merals 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 summarized 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 closures, illustrating once again the category alignment that permeates the design. A85

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:

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

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 evaluation" is in some sense an automatic feature of the two dialects. In addition, in code-generating code such as macros and de-

buggers 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 that designate other semantically significant forms. The intricacies of such "level-crossing" expressions are the stock-in-

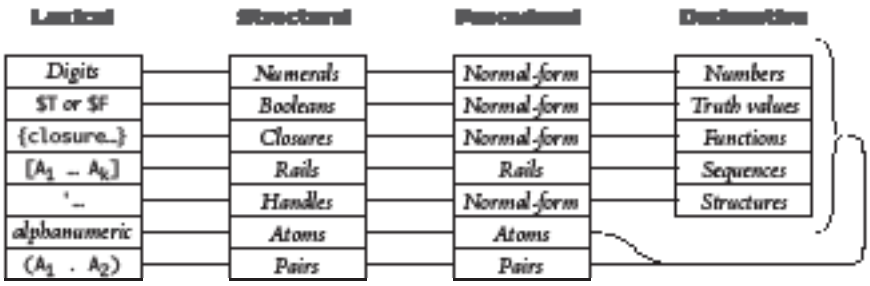

Figure 17 — The Category Structure of 2Lisp (and 3Lisp)

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 discus-

sion 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 Mac- LISP, 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). A86

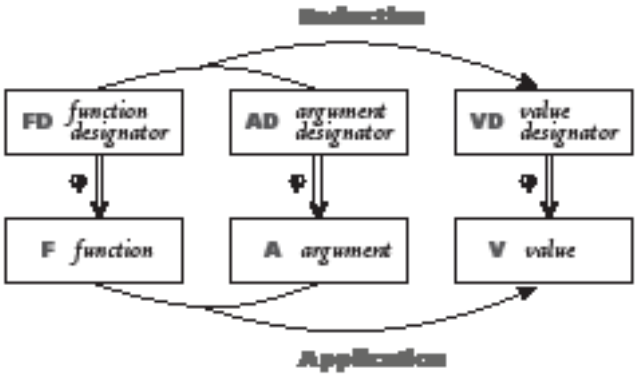

Figure 18

The procedure REDUCE, together with NORMALISE will of

15. Church (1941).

course play a major role in the characterization 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 declar-

ative 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 designators 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.

A87

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).

* * *

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). A88
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 characterization; note that it remains true *even in the case of reflective functions*. A89

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 realize 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 behavior is combined with A90 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 A91
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 A92
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 behavior (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 behaviorally 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 characterization 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 characterization 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.

A93

A94

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 rationalized 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 par-

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). A95

ticular 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 characterized 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.

A96

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). A97 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 recognize 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 behavior 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 analyze 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 problem-solving research (and to programming language research), rather than the particular formalism I have developed and demonstrated by way of illustration. A98

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 rationalized 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 behavior 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] 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. A99

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 A100 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 behavior mimics that of some other world of interest. Under this approach the computational behavior can be taken *in lieu of* or *in place of* the real behavior 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 understand it, underlies Weyhrauch's use of Lisp. A101

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 A102

considerable peril. I in no way rule out computations that in different respects mimic the behavior 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 world being thought about. A103

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. '$F : [A \sqcap B]$' 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:

$$[\, A \,?\, [A \,?\, A] \,?\, [A \,?\, A \,?\, A] \,?\, \ldots\,]$$

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 characterizations 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 computer implementations of 1Lisp, 2Lisp, and 3Lisp developed in the MacLisp and "Lisp Machine" Lisp dialects

A104

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. A105