Reflection and Semantics in LISP

Brian Cantwell Smith*
University of Toronto

1 Abstract

A general architecture is presented, called **procedural reflection**, designed to support self-directed reasoning in a serial programming language. The architecture, illustrated in a revamped dialect of Lisp called **3-Lisp**, involves three steps: (i) reconstructing the semantics of a language so as to deal with both declarative and procedural aspects of program meaning; (ii) embedding a theory of the language—including of its semantics—within the language; and (iii) defining an infinite tower of procedural self-models in terms of this embedded theory, very much like a tower of metacircular interpreters, except causally-connected to each other in a simple but crucial way. In a procedurally reflective architecture, any aspect of process state that can be described in terms of the theory can be rendered explicit, in

[†]© Brian Cantwell Smith 2009 Last edited: December 17, 2009
Draft only (version 0.80) Please do not copy or cite.
Comments welcome brian.cantwell.smith@utoronto.ca
Edited version of a paper first published in the *Conference Record of the Elev-enth Annual acm Symposium on Principles of Programming Languages* (POPL), Salt Lake City, Utah, Jan. 1984, pp. 23–35. The original version was also published as Report No. CSLI-84-8, Stanford University Center for the Study of Language and Information, July 1984.
*Faculty of Information, University of Toronto, 90 Wellesley St W, Toronto, Ontario M5S 1C5 Canada.

structures accessible for program examination and manipulation. Procedural reflection enables a user to define complex programming constructs by writing, within the programming language, direct analogues of those metalinguistic semantical expressions that would normally be used to describe them.

It is argued that the concept of procedural reflection should be added to any language designer's tool kit.

2010 Perspective[α1]

The work reported here, on procedural reflection and 3-Lisp, started out as what I expected to be a small design study—part of a (hopelessly ambitious) project I had undertaken, as a graduate student in the 1970s, to develop a fully reflective knowledge representation system. That project, to have been called **Mantiq**,[α2] never saw the light of day, most pointedly due to my encounter with the fundamental inability of Artificial Intelligence and computer science to deal adequately with the challenges of real-world ontology (the nature of objects, ambiguity and vagueness, relationality and process, etc.). But there were other challenges as well: another goal was to define the Mantiq structural field (effectively: its object or memory system—see p. ■■) at a sufficiently high level of abstraction so as to be able to "fuse" meta-structural and intensional identity, so that *structural* identity could be identified with (and thus used to determine) identity of *meaning*.[α3] The idea was to employ a computationally-intensive background relaxation algorithm to implement the "structural field" (memory system), loosening operational identity criteria to the point that, for example, the Mantiq analogues of (λx,y . x+y) and (λa,b . b+a) would appear to be structurally indistinguishable.

I still think that this issue of intensional identification would be a worthwhile goal to pursue, especially since processing power today would make approximating it more computationally feasible than it was thirty years ago. At any rate, against this background of unrealistic dreams, the 3-Lisp project[α4] was intended as a site to work out the design details of reflection's self-referential structure. In particular, the idea of understanding level-shifting in terms of an idealized unbounded "tower" of referential layers struck me then (and still does now) as at least a good initial idea about the structure of reflection.[α5] So I set out to explore it within the familiar context of Lisp, the "lingua franca" programming language of the MIT Artificial Intelligence Laboratory, where I was enrolled.

1 Introduction

Among programming languages, Lisp is famous for (among other things) providing inchoate self-referential capabilities: standard coding of programs as data structures (s-expressions), a primitive quotation function (QUOTE), explicit access to interpreter procedures (EVAL and APPLY), support for meta-circular interpreters, etc. Yet these capacities have not led to a general understanding of what it is for a computational system to reason, in substantial ways, about its own operations and structures.

There are several reasons we have not developed such an account. First, there is more to reasoning than reference; one also needs a theory, in terms of which to make sense of the referenced domain. A computer system able to reason about itself—what I will call a **reflective** system—will therefore need an account of itself embedded within it. Second, there must be a systematic, causally effective relationship between that embedded account and the system it describes. Without such a connection, the account would be useless—as disconnected as the words of a hapless drunk who carries on about the evils of inebriation, without realising that his story applies to himself. Traditional language embeddings in Lisp (metacircular interpreters and implementations of other languages) are inadequate in just this way; they provide no means for the implicit state of the Lisp process to be reflected, moment by moment, in the explicit terms of the embedded account. Third, a reflective system must be given an appropriate vantage point at which to stand, far enough away to have itself in focus, and yet close enough to see the important details.

This paper presents a general architecture, called **procedural reflection**, to support self-directed reasoning in a serial programming language. The architecture, illustrated in a revamped Lisp dialect called **3-Lisp**, solves all three problems with a single mechanism. The basic idea is to define an infinite tower of procedural self-models, very much like metacircular interpreters,[1] except connected to each other in a simple but critical way. In such an architecture, any aspect of a process' state that can be described in terms of the theory can be rendered explicit, in program accessible structures, at an arbitrary points throughout a computation. Furthermore, as I

1. Steele and Sussman (1978b).

will demonstrate, this apparently infinite architecture can be finitely and efficiently implemented.

The architecture allows the user to define complex programming constructs (such as escape operators, deviant variable passing protocols, and debugging primitives) by writing, within the language, direct analogues of the metalinguistic semantical expressions that would normally be used to describe them. As is always true in semantics, the metatheoretic descriptions must be phrased in terms of some particular set of concepts; in the 3-Lisp case I use a theory based on *environments* and *continuations*. A 3-Lisp program, therefore, at any point during a computation, can easily obtain representations of the environment and continuation characterising the state of the computation at that point. As a result, such constructs as THROW and CATCH, which must otherwise be provided primitively, can be easily defined in 3-Lisp as user procedures (and defined, furthermore, in code that is almost isomorphic to the λ-calculus equations one normally writes, in the metalanguage, to describe such constructs). Moreover, these and other analogous control constructs can be defined without having to write the entire program in a continuation-passing style, of the sort illustrated in Steele (1976).

The point is not to decide at the outset what should and what should not be explicit, in other words (in Steele's example, continuations must be passed around explicitly from the beginning).[a] Rather,

---

a) *(Note: footnotes indicated with letters rather than numerals, and sans-serif font, as in this case, are annotative notes added in 2010, rather than material that appeared in the original paper.)*

This phrasing is somewhat disingenuous, since in a procedurally reflective dialect of the sort presented here the language designer must decide, advance, what aspects of the language *will be able to be made explicit* to user code; those aspects must then be dealt with, explicitly, in the metatheory in terms of which the reflective processor and dialect are themselves defined, and then provided for in the implementation. The original paper would have been better phrased if written as follows: "Although the metatheory (and reflective processor) must deal explicitly with all of those aspects of the language that can, at any point, be made explicit, any user code that does not want to deal with them need not deal with them explicitly. In Steele's dialect, in contrast, in order for an aspect to be referred to explicitly at any point, it must be explicit *throughout the program*. In a sense, therefore, reflection can be understood as providing something like contextual information hiding—or perhaps more

the reflective architecture provides a method of making some aspects of the computation explicit, right in the midst of a computation, even if they were implicit a moment earlier—and in such a way that they can be made implicit once again, a moment later. It provides a mechanism, in other words, when circumstances warrant it, of stepping back, "pulling information out of the sky," dealing with that information appropriately, and then returning into the regular implicit flow of the program.

The thesis on which the 3-Lisp definition rests is the following:

Reflection is simple to build on a semantically sound base. **[R]**

By "semantically sound" I mean more than that the semantics be carefully formulated. Rather, it is assumed throughout that computational structures have a semantic significance that transcends their behavioural import—or, to put this another way, that programs and computational structures are *about* something, over and above the *causal effects* they have on the systems they inhabit. Lisp's `NIL`, for example, evaluates to itself forever—that is its procedural impact. In addition, however, in some contexts—and partially independently—it also *stands for falsehood*. It is that sense of "*meaning false*" that I take to be its declarative import. To be considered "semantically sound," a reconstruction of Lisp semantics must deal explicitly with both of these dimensions of the overall significance of computational structures—both procedural and declarative.[2]

In what follows I will use the phrases "*procedural result*" (or "*what it returns*") to name that to which its effective treatment gives rise, and "*declarative import*" for what a structure designates, declaratively. As well as distinguishing result and import, I will also discriminate

2. This distinction between the procedural and declarative aspects of a program's meaning differs from the traditional distinction in programming language theory between operational and denotational semantics. It is a reconstruction developed within a view that programming languages are properly to be understood in the same theoretical terms used to understand natural language and mind—not just other computer languages.

---

accurately, *contextually-dependent explicitization of otherwise implicit information.*"

The next sentence in the text is more accurate, and more useful.

entities, such as numerals and numbers, that are isomorphic but not identical, if they differ in respect of either import or result.[3] Both distinctions are instances of the general intellectual hygiene of avoiding use/mention errors. Lisp's basic notion of *evaluation*, I will argue, is fundamentally confused on both counts—and should be replaced with independent notions of **designation** and **simplification**. The result will be illustrated in a semantically rationalised dialect, called **2-Lisp**, based on a *simplifying* (designation-preserving) term-reducing processor.

The practical import of thesis [R] is demonstrated in a two-stage argument:

1. The semantically rationalised 2-Lisp is more elegant and theoretically cleaner than any prior Lisp dialect (including both Lisp 1.5 and Scheme); and

2. The reflective dialect 3-Lisp can be very simply defined on top of 2-Lisp—whereas a reflective version of a non-semantically-rationalised Lisp dialect would be inelegant in a spate of ways: gratuitously challenging to design, architecturally baroque, and much more difficult to understand.

The strategy of presenting the general architecture of procedural reflection by developing a concrete instance of it was selected on the grounds that a genuine theory of reflection (perhaps analogous to the theory of recursion) would be difficult to motivate or defend without taking this first, more pragmatic, step. In section 10, however, I will sketch a general "recipe" for adding reflective capabilities to any serial language; 3-Lisp is the result of applying this conversion process to the non-reflective 2-Lisp.

It is sometimes said that there are only a few constructs from which programming languages are assembled—including, for example, predicates, terms, functions, composition, recursion, abstraction, a branching selector, and quantification. Though different from these notions (and not definable in terms of them), reflection is perhaps best viewed as a proposed addition to that family. Given this view, it is helpful to understand reflection by comparing it, in particu-

3. Numerals denote numbers, but (at least in ordinary circumstances) numbers do not denote at all, not being *symbols*.

lar, with recursion—a construct with which it shares many features. Specifically, recursion can seem viciously circular to the uninitiated, and can easily lead to confused implementations if poorly understood. Careful theoretical analysis, however, backed by mathematical theory, underwrites our ability to use recursion in programming languages without doubting its fundamental soundness (in fact, for many programmers, without understanding much about the formal theory at all). Reflective systems, similarly, are initially likely to seem viciously circular (or at least infinite), and are correspondingly difficult to implement without an adequate understanding. The intent of this paper, however, is to argue that reflection is in fact as well-tamed a concept as recursion, and potentially as efficient to use. The long-range goal is not to force programmers to understand the intricacies of designing a reflective dialect, but rather to enable them to use reflection and recursion with equal abandon.

2 Motivating Intuitions

Before taking up technical details, it will help to layout some motivations and assumptions.

By 'reflection' in its most general sense, I mean the ability of an agent to reason not only introspectively, about its self and internal thought processes, but also externally, about its behaviour and situation in the world. Ordinary reasoning is external in a simple sense: most of what we think about (chairs, other people, bank accounts, houses, politics, etc.) is external to us. The point of reflection is to give an agent a more sophisticated stance *from which to consider its own presence in that embedding world*. There is a growing consensus[4] that reflective abilities underlie much of the plasticity with which we deal with the world, both in language (such as when one says "Do you understand what I mean?") and in thought (such as when one wonders how to be compassionate about delivering bad news). Common sense suggests that reflection enables us to master new skills, cope with incomplete knowledge, define terms, examine assumptions, review and distill experiences, learn from unexpected situations, plan, check for consistency, and recover from mistakes.

Although this paper focuses on reflection in programming

4. See Doyle (1980), Weyrauch (1980), Genesereth and Lenat (1980), and Batali (1983).

languages, most of the driving intuitions on which it is based are grounded in considerations of human rationality and language. Tentative steps towards computational reflection, however, are emerging in computational practice, and have also had a motivating impact here. Debugging systems, trace packages, dynamic code optimizers, runtime compilers, macros, metacircular interpreters, error handlers, type declarations, escape operators, comments, and a variety of other programming constructs in one way or another involve structures that refer to or deal with other parts of a computational system. These practices suggest. as a first step towards a more general theory, defining a limited and rather introspective notion of "procedural reflection": self-referential behaviour in procedural languages, in which expressions are primarily used instructionally, to engender behaviour, rather than assertionally, to express judgments or make claims. It is the hope that the lessons learned in this smaller task will serve well in the larger account.[b]

I mentioned at the outset that the general task, in defining a reflective system, is to embed a theory of the system in the system in such a way as to support smooth shifting between reasoning directly about the world and reasoning about that reasoning. Because the subject matter is reasoning, moreover, not merely language, an additional requirement is placed on this embedded theory, also already mentioned, beyond its being descriptive and true: it must also be what I will call **causally connected**, so that the reflective accounts of objects, events and states of affairs are directly tied to those selfsame objects, events and states of affairs. This causal relationship must run both directions: from event to description, and from description back to event. The goal is almost that of creating a magic kingdom, where from a cake you can automatically obtain a recipe, and from a recipe automatically produce a cake.

---

b) In part this is a reference to Mantiq, but I had also planned to develop a next dialect in the series, to be called "4-Lisp," which was to include semantically-rationalized data structures for (external) reference to the real-world, but otherwise to retain 3-Lisp's basic style and control structure. Like Mantiq, 4-Lisp never materialized, due to the challenges of developing representational regimens adequate to real-world ontology.

Existing logical and mathematical cases of self-reference, including both self-referential statements, and models of syntax and proof theory, involve no causation at all, since there is no temporality or behaviour (neither logical nor mathematical systems, per se, *run*). Effective causation is a critical part of any reflective agent, however. As a human example, suppose you were to capsize while canoeing through difficult rapids, and were to swim to shore to figure out what you did wrong. In terms of what I will call "upwards" causal connection, you would need a description of *what you were doing at the moment the mishap occurred*; in the concrete exigencies of that circumstance, merely having a name for yourself, or even a general description of yourself, would be useless. Similarly, in order for your on-shore reflections to be of any subsequent paddling use, you would need "downwards" causal connection as well; no good will come from your merely contemplating a disconnected theory of a wonderfully improved you. As well as stepping back and being able to think about your behaviour, in other words, you must also be able to "step forwards," as it were—to embrace a revised theory of self and "dive back in under it," adjusting your behaviour so as to satisfy the new account. And finally, as already mentioned, when you take the step backwards, to reflect, you need a place to stand that has just the right combination of connection and detachment to make this whole process effective and efficient (it is not an accident that the moment of self-contemplation is like to occur *on shore*).

Reflective computational systems, similarly, must provide both directions of causal connection, and an appropriate vantage point. For example, consider a debugging system that accesses stack frames and other implementation-dependent representations of processor state, in order to give the user an account of what a program is up to in the midst of a computation. Note, first, that stack-frames and implementation byte-codes really are just descriptions, in a rather inelegant language, of the state of the process they describe. Like any description, they make explicit some of what was implicit in the process itself (this is one reason they are useful in debugging). Furthermore, because of the nature of implementation—because, that is, they are constitutively enabling descriptions, not detached observations—they are always available in the implementing code, and always true. They have these properties because they play a causal role in the

very existence of the process they implement, and therefore automatically solve the "reality-to-description" direction of causal connection. Second, debugging systems must solve the "description-to-reality" problem, by providing a way of making revised descriptions of the process true of that process. They carefully provide facilities for altering the underlying state, based on the user's description of what that state should be (i.e., "return from this stack frame immediately"). Without this "map to reality" direction of causal connection, the debugging system, like an abstract model, could have no effect on the process it was examining. And finally, programmers who write debugging systems wrestle with the problem of providing a proper vantage point. In this case, practice has been particularly atheoretical; it is typical to arrange, very cautiously, for the debugger to tiptoe around its own stack frames, in order to avoid control challenges, variable clashes and other unwanted interactions.

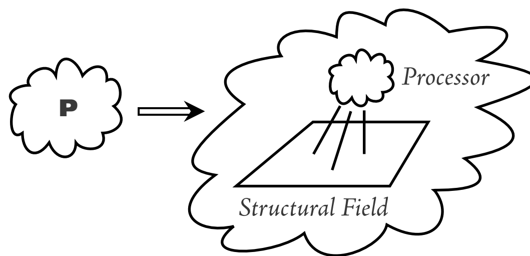

Figure 1 — A Serial Model of Computation

As will be evident in the design of 3-Lisp, all of these concerns can be dealt with in a reflective language in ways that are simple, theoretically elegant, and implementation-independent. The procedural code in the metacircular processor serves as the "theory" discussed above; the causal connection is provided by a mechanism whereby procedures at one level in the reflective tower are run in the process one level above (a clean way, essentially, of enabling a program to define subroutines to be run in its own implementation). In one sense it is all straightforward; the subtlety of 3-Lisp has to do not so much with the power of such a mechanism, which once presented is evident, but with *how such power can be finitely provided*—a question addressed in section 9.

Some final assumptions. I assume a simple serial model of computation, illustrated in figure 1, in which a computational process as a whole is divided into an internal assemblage of program and data structures I will collectively call the **structural field**, coupled with

an internal process that examines and manipulates these structures. In computer science this inner process (or 'homunculus') is typically called the *interpreter*; in order to avoid confusion with semantic notions of interpretation, I will call it the **processor**. While models of reflection for concurrent systems could undoubtedly be formulated, the claim I make here is only that the architecture I will describe is general for calculi of this serial (i.e., single processor) sort.

I will use the term '**structure**' for elements of the structural field, all of which are assumed to be inside the machine; the word will never be used for abstract mathematical or other "external" entities, such as numbers, functions, or radios.[5] Consequently, I call **meta-structural** any structure that designates another structure, reserving **metasyntactic** for *expressions designating linguistic entities or expressions*.[6] Given an interest in internal self-reference, it is clear that both structural field and processor, as well as numbers and functions and the like, must be part of the semantic domain. Note also that the property of being *metastructural* is to be distinguished from the orthogonal property of being *higher-order*, in which terms and arguments may designate functions of any degree (2-Lisp and 3-Lisp will have both properties).[7]

---

5. Although this terminology may be confusing for semanticists who think of a "structure" as a model, I want to avoid calling internal ingredients *expressions*, since the latter term connotes linguistic or notational entities. What I am aiming for is a concept covering both (i) what we would traditionally call data structures, and (ii) the "internal representation" of the program, which we can indirectly use to categorize what we would in ordinary English call the structure of the overall process or agent.
6. Because of the constraints of appropriate causal connection, the meta-structural capability must be provided by primitive quotation mechanisms, as opposed simply to being able to *model* or *designate* syntax—something virtually any calculus can do, using for example Gödel numbering.
7. Most programming languages, such as Fortran and Algol 60, are neither higher-order nor metastructural; the λ-calculus is the former but not the latter, whereas Lisp 1.5 is the latter but not the former (dynamic scoping is a contextual protocol that, coupled with the meta-structural facilities, allows Lisp 1.5 partially to compensate for the fact that it is only first-order. At least some incarnations of Scheme, on the other hand, are both higher-order and metastructural (although Scheme's metastructural powers are expressly limited). As will emerge, 3-Lisp's combination of metastructural and higher-order properties are essential to its reflective capabilities.

3 A Framework for Computational Semantics

Given this background, turn first to questions of semantics. In the simplest case, semantics is taken to involve a mapping, possibly contextually relativized, from a syntactic to semantic domain, as shown in figure 2. The mapping ϕ is typically called an **interpretation function** (to be distinguished, as noted above, from the standard computer science notion of an "interpreter"). Interpretation functions are usually specified inductively, with respect to the compositional structure of the elements of the syntactic domain, which in turn is typically taken to be a set of entities of a syntactic or linguistic sort. Semantic domains may be of any type whatsoever, including domains of behaviour; in reflective systems they will typically include the syntactic and structural domains as proper parts. In this paper, to minimize confusion, I will use a variety of different meta-theoretic variables for different *kinds* of semantic relationship; in the general case, I will use the variable *s* and its cognates (s_1, s_2, s', etc.) to denote symbols or signs, and for any semantic value *d* will say that *s signifies d*, or conversely that *d* is the *significance* or *interpretation* of s.

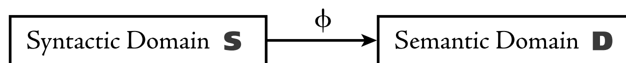

Figure 2 — Simple Semantic Interpretation

It is a fundamental tenet of the proposed approach to reflection to recognize that, in a computational setting, there are several different semantic relationships—not different ways of characterizing one and the same relationship (as operational and denotational semantical accounts are sometimes taken to be, for example), but *genuinely distinct relationships*. These different relationships make for a more complex semantic framework than is standard in logic and model theory, as do ambiguities in the use of words like 'program.' In many settings, such as in purely extensional functional programming languages, such distinctions are relatively inconsequential, and can be harmlessly glossed or elided. But in cases of reflection, self-reference, and metastructural processing, these distinctions, which in other circumstances may seem minor, play a much more important role.

Since the semantical theory adopted to *analyse* 3-Lisp will be at

least partially embedded *within* 3-Lisp, choice of semantical framework affects the formal architecture and design. My approach, therefore, will be to start with basic and simple intuitions, and to identify a finer-grained set of distinctions than are usually employed. I will briefly consider the issue of how the contemporary practice of programming language semantics would be reconstructed in its terms, but the complexities involved in answering that question adequately would take us beyond the scope of the present paper.

Given these preliminaries, I will distinguish three things:

1. The *external* objects and events in the world in which a computational process is embedded—including both real-world objects such as cars and caviar, and set-theoretic abstractions such as numbers and functions (that is: I will adopt a kind of pan-Platonic idealism about mathematical entities);
1. The *internal* elements, structures, or processes inside the computer, including data structures, program representations, execution sequences and so forth (these are all formal objects, in the sense that computation is formal symbol manipulation[c]); and
2. *Notational* or *communicational expressions*, in some externally observable and consensually established medium of interaction, such as strings of characters, streams of words, or sequences of display images on a computer terminal.

The third set—of expressions—are assumed to include the constituents of communication with the computational process (by human agents or other computational processes); the middle set are the ingredients of the process with which those communicating external agents and processes interact; and the first (at least presumptively) are the elements of the world or "subject matter" about which that communication is held. In the human case, the three domains would correspond, respectively, to *world*, *mind*, and *language*.

---

c) Even at the time this paper was published I was critical of the idea that computation could adequately be understood as formal symbol manipulation; I believe that the phrasing "in the sense that" was meant to signal (rather ineffectively) some distancing of my own view from that then-universal assumption. It was not until 1986 that I explicitly argued against such a construal. See «ref "From Symbols to Knowledge", and AOS.»

It is a theoretical truism that the third domain of objects—the elements of communication—are semantic, in the sense of being meaningful, serving as vehicles of meaning, carrying information, or some such. In this work I will take the middle set to be semantic as well—i.e. will assume that *internal structures* are bearers of meaning, information, and/or content. Distinguishing between the semantics of communicative expressions and the semantics of internal structures will be one of the main features of the framework I adopt. It should be noted, however, that in spite of my endorsing the reality of internal structures, and assuming the reality of the embedding world, it is nonetheless true that in the cases I will consider (i.e., ignoring sensors and manipulators), the only things that actually *happen* with computers are communicative interactions. For example, in a case that I might informally describe as "asking my Lisp machine what the square root of two is," what in fact happens, concretely, is that I type an expression such as `(SQRT 2.0)` at the computer, and receive back *some other expression*, probably quite like `1.414`, by way of response. What matters, for our purposes, is that the interaction is carried out entirely in terms of expressions; no structures, numbers, or functions are part of the interactional event (in particular, it is metaphysically precluded, given the presumed philosophy of mathematics, for a computer to *return the square root of two*). The denotation or participation or relevance of any of more abstract objects, such as numbers, must be inferred from, and mediated through, the communicative act.

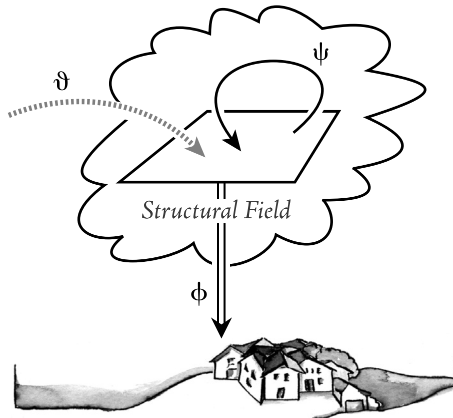

Figure 3 — Semantic Relationships in a Computational Process

I will begin to analyse this complex of relationships using the terminology suggested in figure 3. By ϑ, very simply, I will refer to the relationship between *external notational expressions* and *internal structures*; by 'ψ' I will refer to the processes and behaviours those

structural field elements engender (thus 'ψ' is inherently temporal); and by 'ϕ' I will to the entities in the world that they designate. For mnemonic convenience, relations 'ϕ' and 'ψ' have been named to suggest *philosophy* and *psychology*, respectively, since a study of 'ϕ' is a study of the relationship between structures and the world, whereas a study of 'ψ' is a study of the relationships among symbols, all of which are "within the head" (of person or machine).

Since computation is inherently temporal, the semantic analysis must deal explicitly with relationships across the passage of time. In figure 4, therefore, I have unfolded the diagram of figure 3 across a unit of time, so as to get at a full configuration of these relationships. Entities n_1 and n_2 are intended to be linguistic or communicative entities, as described above;[8] s_1 and s_2 are internal structures over which internal processing is defined. The relationship ϑ, which I will call **internalisation** (and its inverse, ϑ^{-1}, **externalisation**) relates these two kinds of object, as is appropriate given the device or process in question (thus I will say, in addition, that n_1 **notates** s_1). For example, in first order logic n_1 and n_2 would be expressions, written with letters, spaces, '∃' and '∀' signs, etc.; to the extent that s_1 and s_2 could be said to exist, in logic, they would be something like abstract derivation tree types of the corresponding first-order formulae. In Lisp, as we will see, n_1 and n_2 would be the input and output expressions, written with letters and parentheses, or perhaps with boxes and arrows; s_1 and s_2 would be the corresponding cons cells in the s-expression heap.

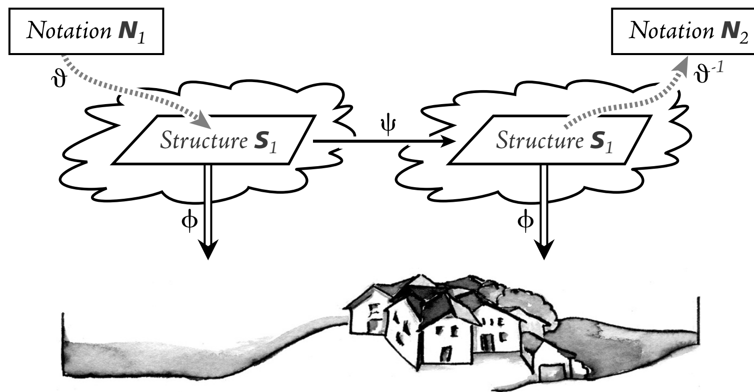

Figure 4 — A Framework/or Computational Semantics

8. That is: the variable 'n_1' and its cognates are used in this text is as a meta-level variable to denote a linguistic or communication expression; etc.

In contrast, d_1 and d_2 are elements or fragments of the embedding world, and ϕ is the relationship that internal structures bear to them. ϕ, in other words, is semantics' so-called "interpretation function" that makes explicit what I will call the **designation** of internal structures (not, note, the designation of linguistic expressions or terms, which would be described by $\vartheta \circ \phi$). The relationship between my mental token representing T. S. Eliot, for example, and the poet himself, would be formulated as part of ϕ, whereas the relationship between the public name 'T. S. Eliot' and the poet would be expressed as $\phi(\vartheta($"T. S. Eliot"$))$ = T. S. Eliot. Similarly, ϕ would relate an internal "numeral" structure (say, the numeral 3) to the corresponding number—if I can be permitted to use the word 'numeral' to refer to internal structures as well as to external expressions. As mentioned at the outset, my focus on ϕ is evidence of the permeating semantical assumption that all structures have designations—or, to put it another way, that in the computational realm I am considering, all structures are taken to be symbols.[9]

In contrast to ϑ and ϕ, the relation ψ always (and necessarily, since it does not have access to anything else) relates some internal structures to others, or to behaviours over them. To the extent that it would make sense to talk of a ψ in logic, it would be approximately the formally computed derivability relationship ($\vdash$); in a natural deduction or resolution schemes, ψ would be a subset of the derivability relationship, picking out the particular inference procedures those regimens adopt. In a computational setting, however, ψ would be the function computed by the processor (i.e., in traditional Lisp it is *evaluation*).

9. For what I might call *declarative languages*, there is a natural account of the relationship between linguistic expressions and in-the-world designations that need not make crucial reference to issues of processing (to which I will turn in a moment). It is for such languages, in particular, that the composition $\phi \circ \vartheta$ (call it ϕ'), would be formulated. For obvious reasons, it is ϕ' that is typically studied in mathematical model theory and logic, since those fields do not deal in any crucial way with the *active use* of the languages they study. In logic, for example, ϕ' would be the interpretation function of standard model theory. In what I will call *computational* languages, on the other hand, questions of processing (ψ) do arise for all aspects of significance—and so, in a vaguely Wittgensteinian sense, ϕ' cannot in general be explicated independent of ψ.

The relationships ϑ, ψ, and ϕ have different relative importance in different semiotic disciplines, and relationships among them have been given different names. For example, ϑ is usually ignored in logic, and there is little tendency to view the study of ψ, called proof theory, as *semantical*, although it is always related to semantics, as in proving soundness and completeness.[10] In addition, there are a variety of "independence" claims that have arisen in different fields. That ψ does not uniquely determine ϕ, for example, is the "psychology narrowly construed" and concomitant methodological solipsism of Putnam, Fodor, and others.[11] That ϑ is usually specifiable compositionally and independently of ψ or ϕ is essentially a statement of the autonomy thesis for language. Similarly, when ϑ cannot be specified independently of ψ, computer science will say that a programming language "cannot be parsed except at runtime" (a property exemplified by Teco and the first versions of Smalltalk[12]).

A thorough analysis of these semantic relationships, however, and of the relationships among them, is the subject of a different paper. For present purposes I need not take a stand on which of ϑ, ψ, or ϕ has a prior claim on being "semantics," but it will help to have some English terminology for some of these relations, in order not to have to devolve into formalism. For discussion, therefore, I will refer to the "ϕ" of a structure as its **declarative import**, and to its "ψ" as its **procedural consequence**.[d] It is also convenient to identify some of the situations when two of the six entities (n_1, n_2, s_1, s_2, d_1 and d_2) are identical. In particular, I will say that s_1 is **self-referential** if

10. Soundness and completeness can be expressed as $\psi(s_1,s_1) \equiv [\phi(s_1) \subseteq \phi(d_1)]$, if one takes ψ to be a relation, and ϕ to be an inverse satisfaction relationship between sentences and possible worlds that satisfy them.

11. See Fodor (1980).

12. Teco ("text editor and corrector") was a string-processing language which ran on the "Incompatible Time Sharing Systems" (ITS) at the MIT Artificial Intelligence Lab in the 1970s. It is now remembered primarily as the programming language in which the initial versions of the still-popular text editor EMACS were written. Smalltalk, an object-centered, dynamically-typed, "reflective" programming language, was developed at the Xerox Palo Alto Research Center (PARC) by Alan Kay and his colleagues, also during the 1970s.

d) «This was already said. Check that—but also check all the terminology used for these relations; there is redundancy and confusion throughout.»

$s_1 = d_1$, that ψ **de-references** s_1 if $s_2 = d_1$, and that ψ is **designation-preserving** (at s_1) when $d_1 = d_2$ (as it always is, for example, in the λ-calculus, where at least in the standard model ψ—some combination of α and β-reduction—does not alter the interpretation).

It is natural to ask what a *program* is, what *programming language semantics* gives an account of, and how (this is a related question) φ and ψ relate in the programming language case. An adequate answer to this, however, introduces a maze of complexity that I will have to defer to future work. To appreciate some of the difficulties, note that there are two different ways in which we can conceive of a program, suggesting different semantical analyses.[e] On the one hand, a program can be viewed as a linguistic object that *describes* or *signifies* a computational process consisting of the data structures and activities that result from (or arise during) its execution. In this sense a program is primarily a *referential* or *communicative* entity—not so much playing a causal role within a computational process so much as existing outside the process and representing it. Putting aside for a moment the question of whom it is meant to communicate to, I would simply that on such a reading a program is in the domain of ϑ, and, roughly, that $\phi \circ \vartheta$ of such an expression would be the computation described. The same characterization would, of course, apply to a specification; indeed, the only salient difference might be that a specification would avoid using non-effective concepts in describing behaviour. One would expect specifications to be stated in a declarative language (in the sense defined in footnote ■■), since specifications are not, per se, intended to be executed or run, even though they speak about behaviours or computations. Thus, for program or specification *b* describing computational process *c*, we would have (for the relevant language) something like $\phi(\vartheta(b))=c$. If *b* were a *program*, there would be an additional constraint that the program somehow play a causal role in engendering the computational process *c* that it is taken to describe.

There is an alternative conception, however, which places the program *inside* the machine, as a causal participant in the behav-

e) «This may be the first occurrence of my on-going attention to the differences between and among *specificational*, *ingrediential*, and *communicational* views of programs. Refer back to the 2010 perspective at the outset; and forward to the places where I have the pictures, etc.»

iour that results. This view is closer to the one implicitly adopted in figure 1, and I believe that it is closer to the way in which a Lisp program *must be semantically analysed if we are to understand Lisp's emergent reflective properties*. In some ways this different view has a von Neumann character, in the sense of equating program and data. On this view, the more appropriate equation would seem to be $\psi(\phi(b))=c$, since one would expect the processing of the program to yield the appropriate behaviour. One would seem to have to reconcile this equation with that in the previous paragraph, although it is not clear that this would be possible.[f]

Disentangling these points will require further work. What I can say here is that programming language semantics seems to focus on what, in the terminology I am using, would seem be an amalgam of ψ and ϕ. For our purposes I need only note that we will have to keep ψ and ϕ *strictly separate*, while recognising (because of context relativity and non-local effects) that just because they are distinct does not mean they are independent. Formally, I would need to specify a general significance function Σ,[13] which recursively specifies ϕ and ψ together. In particular, given any structure s_1, and any state of the processor and the rest of the field (encoded, say, in an environment, continuation, and perhaps a store), Σ will specify the structure, configuration, and state that would result (i.e., it will specify the *use* of s_1), and also the signifying relationship that s_1 bears to the world. For example, given a Lisp structure of the form `(+ 1 (PROG (SETQ A 2) A))`, Σ would specify that the whole structure designated the number three, that it would "return" (i.e., that its procedural consequence would be) the numeral `3`, and that the machine would be left in a state in which the binding of the variable `A` was changed to the (structural) numeral `2`.[g]

---

13. This is what was done in «ref TR».

f) «I believe this last sentence is either confused or wrong. Think about it and fix as appropriate.»

g) Computer science talks about a variable being "bound to" something—namely, to its *value*—though, as evident in the semantical reconstruction being carried out here, that usually means to a co-referential structure. Strictly speaking, that is, a programming language variable would be bound to a *numeral*, not to a *number*—and should be so described, in contexts in which the differences between numerals and numbers are significant. In mathemat-

Before leaving semantics completely, it is instructive to apply these various distinctions to traditional Lisp. I said above that all interaction with computational processes is mediated by communication; this can be stated in the present terminology by noting that ϑ and ϑ^{-1} (internalization and externalization) are a part of any interaction. Thus Lisp's "READ-EVAL-PRINT" loop is mirrored in this analysis as an iterated version of $\vartheta^{-1} \circ \psi \circ \vartheta$ (i.e., if n_1 is an expression that you type as input to a Lisp system, returning n_2 as output, then $n_2 = \vartheta^{-1}(\psi(\vartheta(n_1)))$. The Lisp structural field, as it happens, has an extremely simple compositional structure, based on a binary directed graph of atomic elements called *cons-cells*, extended with atoms, numerals, and so forth. The linguistic or communicative expressions that we use to represent Lisp programs—the formal language objects that we edit with our editors and print in books and on terminal screens—is a separate lexical (or sometimes graphical) entity with its own syntax (parentheses and identifiers in the lexical case; boxes and arrows in the graphical).

In Lisp there is a relatively close correspondence between expressions and structures; it is one-to-one in the graphical case, but the standard lexical notation is both ambiguous (because of shared tails) and incomplete (because of its inability to represent cyclical structures). The correspondence need not have been as close as it is; the process of converting from external syntax or notation to internal structure could involve arbitrary amounts of computation, as evidenced by read macros and other syntactic or notational devices. But the important point is that it is *structural field elements*, not *notations*, over which most Lisp operations are defined. If you type "(RPLACA '(A . B) 'C)", for example, the processor will (as expected) first create and then change the CAR (first element) of a field structure;

---

ics and logic, variables are likely, if bound to anything, to be bound to *numbers*—i.e., to what is here being called declarative import. Moreover, it is also more common in logic and mathematics to describe a variable as "bound *by*" something—namely, bound by *quantifiers*, *scoping constructs*, etc. This is just one small instance of the general phenomenon of computer science's using, as technical terminology, vocabulary and phrasings derived from logic, but in its own distinct ways. Sometimes, as here, the differences are subtle, and not usually distracting; sometimes, as with the word 'semantics,' they are major, and cause of considerable confusion. See AOS.

it will not back up your terminal and erase the eleventh character of the *expression* that you typed as input (if that were even physically possible). Similarly, Lisp atoms are field elements, not to be confused with their lexical representations (sometimes called 'P-names' or "print-names"). Again, quoted forms such as (QUOTE ABC) designate structural field elements, not input strings. The form (QUOTE ___), in other words, is a *structural* quotation operator; *notational* quotation is different, usually notated with string quotes (as in "ABC").[14]

4 Evaluation Considered Harmful[h]

The claim that all three relationships (ϑ, ϕ, and ψ) figure crucially in an account of Lisp is not a formal one. It makes an empirical claim on the minds of programmers, and cannot be settled by pointing to any current theories or implementations. Arguments in its behalf would point to the fact that Lisp's numerals are universally taken to designate numbers, and that the atoms T and NIL (at least in predicative contexts) are similarly understood to stand for truth and falsity—no one could learn Lisp without learning these facts, and the behaviour of Lisp systems is only intelligible on such an assumption.[i] In what follows I will therefore state, without qualification, that '3' (i.e., the

14. The string "(QUOTE ABC)" notates a structure that designates another structure that in turn could be notated with the string "ABC". The string " "ABC" ", on the other hand, notates a structure that designates the string "ABC" directly.

h) This section title is a play on Edsger W. Dijkstra's legendary "GO TO Statement Considered Harmful" (*Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147–48). No computer scientist in the 1980s would have failed to recognize the illusion; the *Communications of the ACM* (Association for Computing Machinery) was the première professional computer science journal at the time, and Dijkstra's letter was widely taken to have inaugurated serious theoretical analysis of programming. Cf. this note from the History of Computing Project:

> "In 1968 Edsger Dijkstra laid the foundation stone in the march towards creating structure in the domain of programming by writing, not a scholarly paper on the subject, but instead a letter to the editor entitled "GO TO Statement Considered Harmful". (*Comm. ACM*, August 1968) The movement to develop reliable software was underway."

See http://www.thocp.net/biographies/dijkstra_edsger.htm

i) «Put in a pointer to (and discussion of) the "normatively governed effective mechanism" construal of logic and other intentional systems in other papers.»

structural numeral notated by the string character "3") designates three; that T designates truth, that (EQ 'A 'B) designates falsity, etc. In a similar spirit, I will claim that the structure (CAR '(A . B)) *designates* the atom A; this is manifested by the fact that people, in describing Lisp, use expressions such as "If the CAR of the list is LAMBDA, then it is a procedure," where the ingredient term "the CAR of the list" is used as an *English referring expression*—specifically as a *singular term*—not as a quoted fragment of Lisp (and English, or natural language generally, is by definition the locus of what designation is). (QUOTE A), or 'A, is another way of *naming* or *designating* the atom A; that is just what quotation is. By the same token, I will take such atoms as CAR and + to name or designate the obvious corresponding functions.

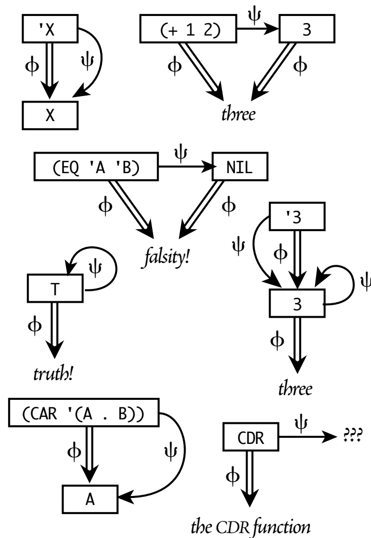

Figure 5 — Lisp Evaluation vs. Designation: Some Examples

What, then, is the relationship between the declarative import (ϕ) of Lisp structures and their procedural consequence (ψ)? Inspection of the superficially rather bewildering data given in figure 5 shows that Lisp obeys the following constraint, where S is the domain of structural field elements (more must be said about ψ in those cases where $\phi(\psi(s)) = \phi(s)$, since the identity function would satisfy this equation):

$$\forall s \in S \text{ if } \phi(s) \in S \text{ then } \psi(s) = \phi(s) \quad \text{else } \phi(\psi(s)) = \phi(s) \qquad [1]$$

All Lisps, including Scheme,[15] in other words, *de-reference* any structure whose designation is another structure, but will return a *co-designating* structure for any whose designation is external to the machine. This regularity, which generates the variety of cases illustrated in figure 5, is depicted in figure 6. Whereas evaluation is often thought to correspond to the semantic interpretation function ϕ, in other words, and therefore to have type *expressions* → *values*, evalua-

15. Steele and Sussman (1978a).

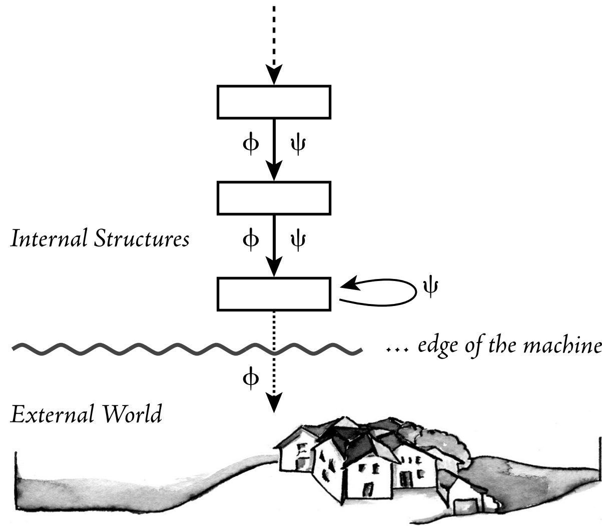

Figure 6 — Lisp's "De-reference if You Can" Evaluation Protocol

tion in Lisp is often a *designation-preserving* operation. In fact, it is a metaphysical fact that no computer can evaluate a structure such as `(+ 2 3)`, if that means "returning what is designated," at least on the Platonist understanding of number I am working with, any more than it can evaluate the name *Hesperus*, or than it is likely to be able to evaluate the name *peanut butter*.

I take it as self-evident that obeying equation [1] is anomalous. It implies, among other things, that even if in a case in which one knows what y is, and knows that x evaluates to y, one still does not know what x designates. It also licenses such semantic anomalies as `(+ 1 '2)`, which—contrary, I would argue, both to common and to theoretical sense—will evaluate to (the structure!) `3` in all extant Lisps. Informally, I will say that Lisp's evaluator *crosses semantical levels*, and therefore obscures the difference between simplification and designation. Given that processors cannot always de-reference (since by assumption the co-domain is limited to the structural field), the only semantically consistent non-level-crossing behaviour they can exhibit in general is to preserve designation. It seems, therefore, that they should always *simplify*, and therefore obey the following constraint (diagrammed in figure 7):

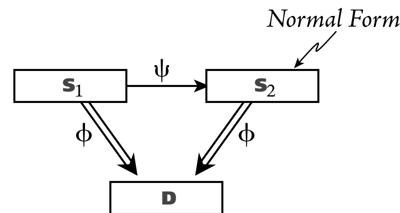

Figure 7 — A Normalisation Protocol

$$\forall s \in S \;\; \phi(\psi(s)) = \phi(s) \wedge \text{normal-form}(\psi(s)) \qquad [2]$$

The content of this equation clearly depends entirely on the content of the predicate "normal-form" (if "normal-form" were l*x*.true, then ψ could be the identity function). In the λ-calculus, the notion of normal-formedness is defined in terms of the processing protocols (α- and β-reduction), but I cannot use any such definition here, on threat of circularity. Instead, I will say that a structure is in normal form if and only if it satisfies the following three independent conditions:

1. It is **context-independent**, in the sense of having the same declarative (ϕ) and procedural (ψ) import independent of the context of use;
2. It is **side-effect-free**, implying that the processing of the structure will have no effect on the structural field, processor state, or external world; and
3. It is **stable**, meaning that it must simplify to itself in all contexts, so that ψ will be idempotent.

We would then have to prove, given a language specification, that equation [2] is satisfied (as it is in the case of 2-Lisp and 3-Lisp)

Two notes. First, I will not use the terms 'evaluate' or 'value' for expressions or structures, referring instead to **normalisation** for ψ, and **designation** for ϕ. I will sometimes call the result of normalising a structure its *result* or what it *returns*. There is also a problem with the terms 'apply' and 'application.' In standard Lisps, APPLY is (the name of) a function from structures and arguments onto values, but like 'evaluate', its use is rife with use/mention confusions. As illustrated in figure 8, I will use 'apply' for mathematical function application—i.e., to refer to a relation-

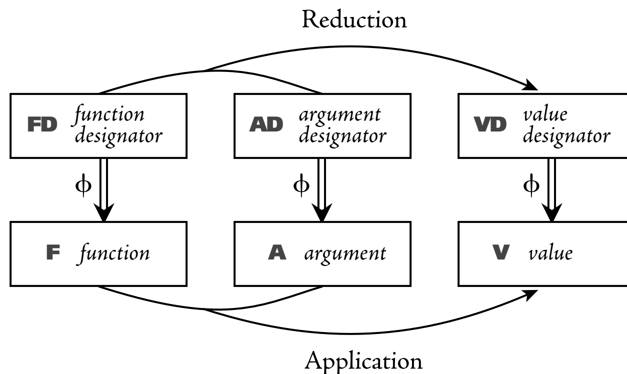

Figure 8 —Application vs. Reduction

ship between a function, some arguments, and the value of the function applied to those arguments—and the term '**reduce**' to relate the three structures that *designate* functions, arguments, and values, respectively. Note that this terminological practice retains use of the term 'value' (as, for example, in the previous sentence), but only to name that entity onto which a mathematical function maps its arguments.

Second, the idea of a normalising processor depends on the idea that symbolic structures have a semantic significance *prior to, and independent of*, the way in which they are treated by the processor.[j] Without this assumption we could not even *ask* about the semantic character of the Lisp (or any other) processor, let alone suggest a cleaner version. Without such an assumption, more generally, one cannot say that a given processor is correct, or coherent, or incoherent; it would merely be what it is. Given one account of what it did (such as an implementation), one could compare that to another account (such as a specification). One could also prove that it had certain properties, such as that it always terminated, or that it used resources in certain ways. One could even prove properties of programs written in the language it runs (from a specification of the ALGOL processor, for example, one might prove that a particular program sorted its input). However, none of these questions deal with the question I am taking to be more fundamental: about the semantical nature of the processor itself. I am not satisfied to say that the semantics of `(CAR '(A . B))` is `A` *because that is how the processor is defined*; rather, I want to say that the processor was defined that way *because* `A` *is what* `(CAR I (A . B))` *designates*. Semantics, in other words, should be a tool with which to *judge* systems, not merely a method of *describing* them.

5 2·Lisp: A Semantically Rationalised Dialect

Having torn apart the notion of evaluation into two constituent notions (designation and simplification), we need to start at the beginning, and build Lisp over again. What I am calling **2-Lisp** is a proposed result. Some summary comments can be made.

---

j) «Talk about this in relation to Amala, Mike Dixon's thesis, errors in the definition of factorial, etc.—and to subsequent semantical inquiry (also to logic).»

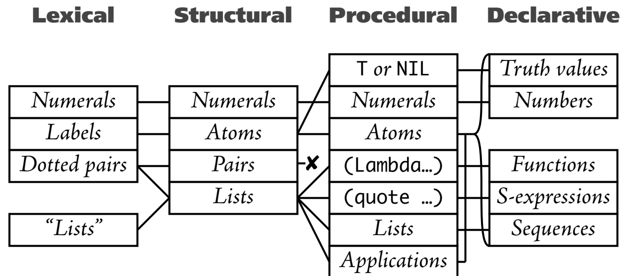

Figure 9 — The Category Structure of Lisp 1.5

First, I have reconstructed what I call the **category structure** of Lisp, requiring that the categories into which Lisp structures are sorted, for various purposes, "line up" (giving the dialect a property I will call **category alignment**). More specifically, Lisp expressions are sorted into categories by *notation*, *structure* (atoms, cons pairs, numerals), *procedural treatment* (the "dispatch" inside the traditional `EVAL`), and *declarative semantics* (the type of object designated). As illustrated in figure 9, these categories are traditionally not aligned; *lists*, a derived structure type, include some of the pairs and one atom (`NIL`); the procedural regimen (ψ) treats some pairs (those with `LAMBDA` in the `CAR`) in one way, most atoms (except `T` and `NIL`) in another, and so forth. In 2-Lisp, in contrast, I have required the notational, structural, procedural, and semantic categories to correspond, as much as practicable, one-to-one, as illustrated in figure 10 (this is a bit of an oversimplification, since atoms and pairs—representing arbitrary variables and arbitrary function application structures or redexes—can designate entities of any semantic type).

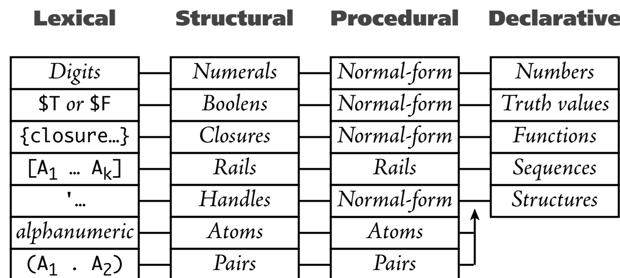

Figure 10 — The Category Structure of 2-Lisp and 3-Lisp

2-Lisp is summarized in the sidebar ("An Overview of 2-Lisp," starting below); some additional comments can be made here. Like most mathematical and logical languages, 2-Lisp is almost entirely "**declaratively extensional**". Thus `(+ 1 2)`, an abbreviation for `(+ . [1 2])`, designates the value of the application of the function designated by the atom `+` to the sequence of numbers designated by the rail `[1 2]`. In other words, `(+ 1 2)` designates the number three, of which the numeral `3` is the normal-form designator; `(+ 1 2)` therefore normalises to the numeral `3`, as expected. 2-Lisp is also usually call-by-value (what we might call "**procedurally extensional**"), in the sense that procedures by and large normalise their arguments. Thus the structure `(+ 1 (BLOCK (PRINT "HELLO") 2)` will normalise to `3`, printing out `"HELLO"` in the process.

Many properties of Lisp that must normally be posited in an ad hoc way fall out directly from this analysis. For example, it normally requires explicit statement that some atoms, such as `T` and `NIL` and

An Overview of 2-Lisp

Begin with objects. Ignoring input/output categories such as *characters*, *strings*, and *streams*, there are seven 2-Lisp structure types, as illustrated in Table 1. The **numerals** (notated as usual) and the two **Boolean constants** (notated '`$T`' and '`$F`') are unique (i.e., canonical), atomic, normal-form designators of numbers and truth-values, respectively. Rails (notated '`[A`$_1$ `A`$_2$ … `A`$_k$`]`') designate sequences; they resemble standard Lisp lists, but are distinguished from pairs in order to avoid category confusion, and are given their own name in order to avoid confusion with *sequences*, *vectors*, and *tuples*, all of which are normally taken to be Platonic ideals.

All **atoms** are used as variables (i.e., as context-dependent names); as a consequence, no atom is normal-form, and no atom will ever be returned as the result of processing a structure (although a designator of an atom may be). **Pairs** (sometimes also called **redexes**—notated '`(A`$_1$ `. A`$_2$`)`'—designate the value of the function designated by their `CAR` (i.e., `A`$_1$) applied to the arguments designated by their `CDR` (`A`$_2$). By taking notational form '`(A`$_1$ `A`$_2$ … `A`$_k$`)`' to abbreviate '`(A`$_1$ `. [A`$_2$ … `A`$_k$`])`' instead of Lisp's traditional '`(A`$_1$ `. (A`$_2$ `. … (A`$_k$ `. NIL)…)))`', we preserve the standard look of Lisp programs, without sacrificing category alignment. (Note that 2-Lisp has no distinguished atom `NIL`, and '`()`' is a notational error—corresponding to no structural field element.) **Closures** (no-

all numerals, are self-evaluating; in 2-Lisp, the fact that the Boolean constants are self-normalising follows directly from the fact that they are normal-form designators. Similarly, closures are a natural category, and distinguishable from the functions they designate (there is ambiguity, in Scheme, as to whether the value of + is a function or a closure). Finally, because of category alignment, if X designates a sequence of the first three numbers (i.e., it is bound to the rail [1 2 3]), then (+ . X) will designate the number five and normalise to the numeral 5; no metatheoretic machinery is needed for this "un-currying" operation (in regular Lisps one must use (APPLY '+ X); in Scheme, (APPLY + X)).

Numerous properties of 2-Lisp will be ignored in this paper. The dialect is defined in Smith (1982) to include side-effects, intensional procedures (procedures which do not normalise their arguments), and a variety of other sometimes-shunned properties, in part to show that this semantic reconstruction being argued for here is

An Overview of 2-Lisp (cont'd)

tated '{CLOSURE: … }') are normal-form function designators, but they are not canonical, since it is not in general decidable whether two structures designate the same function. Finally, **handles** are unique normal-form designators of all structures; they are notated with a leading single quote mark (thus 'A notates the handle of the atom notated A, and '(A . B) notates the handle of the pair notated (A . B), etc. Because designation and simplification are orthogonal, quotation is a structural primitive, not a special procedure (although QUOTE is easy to define as a user function in 3-Lisp).

Turn next to the functions (and use '⇒' to mean 'normalises to'). There are the usual arithmetic primitives (+, -, *, and /). Identity (signified with '=') is computable over the full semantic domain except functions; thus (= 3 (+ 1 2)) ⇒ $T, but (= + (LAMBDA [X] (+ X X))) will generate a processing error, even though it designates truth. The traditionally rather atheoretical difference between EQ and EQUAL turns out to be an expected difference in granularity between the identity of mathematical sequences and their syntactic designators; thus:†

```
(=  [1 2 3J  [1 2 3])   ⇒ $T
(= '[1 2 3] '[1 2 3])   ⇒ $F
(=  [1 2 3J '[1 2 3])   ⇒ $F
```

1ST and REST are the CAR/CDR analogues on both sequences and rails (i.e.,

compatible with the full gamut of features found in real programming languages. Recursion is defined with respect to an analysis using explicit fixed-point operators. 2-Lisp is an eminently usable dialect (it subsumes Scheme but is more powerful, in part because of the metastructural access to closures), although it is ruthlessly semantically strict.

6 Self-Reference in 2·Lisp

Turn now to matters of self-reference.

Traditional Lisps provide names (`EVAL` and `APPLY`) for the primitive processor procedures; the 2-Lisp analogues are `NORMALISE` and `REDUCE`. Ignoring for a moment context arguments such as environments. and continuations, `(NORMALISE '(+ 2 3))` designates the normal-form structure to which `(+ 2 3)` normalises, and therefore returns the handle `'5`. Similarly:

<table>
<tr><th>Type</th><th>Numerals</th><th>Booleans</th><th>Handles</th><th>Closures</th><th>Rails</th><th>Atoms</th><th>Pairs</th></tr>
<tr><td>Designation</td><td>Numbers</td><td>Truth values</td><td>Structures</td><td>Functions</td><td>Sequences</td><td>(ϕ of bndg)</td><td>(value of app)</td></tr>
<tr><td>Normal</td><td colspan="4">Yes</td><td>Some</td><td colspan="2">No</td></tr>
<tr><td>Canonical</td><td colspan="3">Yes</td><td colspan="2">No</td><td colspan="2">N/A</td></tr>
<tr><td>Constructor</td><td colspan="3">N/A</td><td>CCONS</td><td>RCONS</td><td>ACONS</td><td>PCONS</td></tr>
<tr><td>Notation</td><td>Digits</td><td>$T or $F</td><td>'structure</td><td>{closure …}</td><td>[s_1 … s_2]</td><td>Alphanumerics</td><td>(s_1 . s_2)</td></tr>
</table>

Table 1 — The 2-Lisp (and 3-Lisp) categories

have overloaded definitions); thus `(1ST [10 20 30])` ⇒ `10`; and `(REST [10 20 30])` ⇒ `[20 30]`. `CAR` and `CDR` are defined over pairs; thus `(CAR '(A . B))` ⇒ `'A` (because it designates `A`), and `(CDR '(+ 1 2))` ⇒ `'[1 2]`. The pair constructor is called `PCONS` (thus `(PCONS 'A 'B)` ⇒ `'(A . B)`; the corresponding constructors for atoms, rails, and closures are `ACONS`, `RCONS`, and `CCONS`, respectively. There are eleven primitive characteristic predicates—seven for the internal structural types (`ATOM`, `PAIR`, `RAIL`, `BOOLEAN`, `NUMERAL`, `CLOSURE`, and `HANDLE`) and four for the external types (`NUMBER`, `TRUTH-VALUE`, `SEQUENCE`, and `FUNCTION`). Thus:

```
(NORMALISE '(CAR '(A. B)))       ⇒ ''A
(NORMALISE (PCONS '= '[2 3J))  ⇒ '$F
(REDUCE '1ST '[10 20 30J)       ⇒ '10
```

More generally—and entirely intuitively—the basic idea is that ϕ(NORMALISE) = ψ, to be contrasted with $\phi(\downarrow)$, which is approximately ϕ, except that because $\downarrow$ is a partial function we have $\phi(\downarrow \circ$ NORMALISE$) = \phi$. Given these equations, the behaviour illustrated in the foregoing examples is forced by general semantical considerations.

In any computational formalism able to model its own syntax and structures,[16] it is possible to construct what are commonly known

16. Virtually any language has the requisite power to do this kind of modelling. In a language with metastructural abilities, the metacircular processor can represent programs for the MCP as themselves—this is always done

An Overview of 2-Lisp (cont'd)

```
(NUMBER 3)    ⇒ $T
(NUMERAL '3)  ⇒ $T
(NUMBER '3)   ⇒ $F
(FUNCTION +)  ⇒ $T
(FUNCTION '+) ⇒ $F
```

Procedurally intensional IF and COND are defined as usual; BLOCK (as in Scheme) is like standard Lisp's PROGN. BODY, PATTERN, and ENVIRONMENT are the three selector functions on closures. Finally, functions are usually "defined" (i.e., conveniently designated in a contextually relative way) with structures of the form (LAMBDA SIMPLE ARGS BODY) (the term SIMPLE will be explained presently); thus (LAMBDA SIMPLE [X] (+ X X)) normalises to a closure that designates a function that doubles numbers:

```
((LAMBDA SIMPLE [X] (+ X X)) 4)   ⇒ 8
```

2-Lisp is *higher-order*, and therefore lexically scoped, like the λ-calculus and Scheme. As mentioned earlier, however, and illustrated with the handles in the previous paragraph, it is also *metastructural*, providing an explicit ability to name internal structures. Two primitive procedures, called UP and DOWN (usually abbreviated '↑' and '↓', respectively) help to mediate this metastruc-

as *metacircular interpreters*, which I will call **metacircular processors** (or MCPS)—"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 in a prior, independently-understood language—but rather "define" the processor only in terms of itself. This circularity takes two forms. First, on the procedural side, MCPS must be *run* by the processor in order to yield any sort of behaviour (strictly speaking, that is, MCPS

in Lisp MCPS—but we need not define that to be an essential property. The term 'metacircular processor' is by no means strictly defined; there are various constraints that one might or might not put on it. My general approach has been to view as metacircular any *non-causally connected* model of a calculus within itself; thus the 3-Lisp reflective processor is *not* meta-circular, by my lights, because it *does* have the requisite causal connections, and is therefore an essential (not additional) part of the 3-Lisp architecture.

ıy (there is otherwise no way to add or remove quotes— '2 will
'2 forever, never to 2. Specifically, ↑STRUC designates the normal-
tor of the designation of STRUC; i.e., ↑STRUC designates what STRUC
› (therefore ↑(+ 2 3) ⇒ '5). Thus (note that '↑' is call-by-value but
/ely extensional):

AMBDA SIMPLE [X] X) — designates a function
AMBDA SIMPLE [X] X) — designates a pair or redex
.AMBDA SIMPLE [X] X) — designates a closure

RUC designates the designation of the designation of STRUC, pro-
ıe designation of STRUC is in normal-form (therefore ↓'2 ⇒ 2).
/ays equivalent to STRUC, in terms of both designation and result;
when it is defined. Thus if DOUBLE is bound to (the result of
(LAMBDA [X] (+ X X)), then (BODY DOUBLE) generates an error, since
sional and DOUBLE designates a function, but (BODY ↑DOUBLE) will
pair (+ X X).

e one structure designates a sequence and one a rail.

are *programs*, not *processors*). Second, the behaviour they would thereby engender—which is to say, the behaviour they must also therefore designate—can be discerned from them only if one knows beforehand what that behaviour is (i.e., what the processor does).[17] Nonetheless, such processors are pedagogically illuminating, and play a critical role in the development of procedural reflection.

The role of MCPs is illustrated in figure II, showing how, if we ever replace P in figure I with a process that results from P processing the metacircular processor MCP, it would still correctly engender the behaviour of any overall program. Taking processes to be functions from structures onto behaviour, therefore (whatever behaviour is—functions from initial to final states, say), and calling the primitive processor P, we should be able to prove that P(MCP) ≈ P, where by '≈' we mean *behaviourally equivalent in some appropriate sense*. The equivalence, of course, is in a certain sense global, or at the level of types; by and large the primitive processor and the processor resulting from the explicit running of the MCP cannot be arbitrarily mixed. If a variable is bound by the underlying processor P, it will not be able to be looked up by the metacircular code, for example. Similarly, if the metacircular processor encounters a control-structure primitive, such as a THROW or a QUIT, it will not cause the metacircular processor itself to exit prematurely, or to terminate. Rather, the point is that if an entire computation is run by the process that results from the explicit processing of the MCP by P, the results will be the same (modulo time) as if that entire computation had been carried out by P directly. MCPs, to put this in language to be used in providing

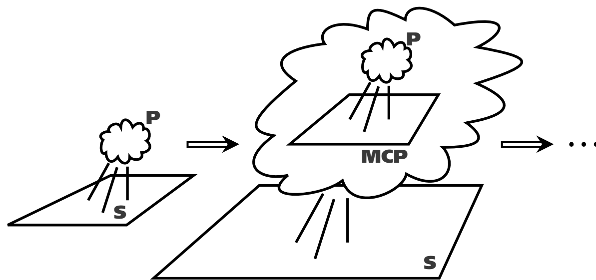

Figure II — Meta-Circular Processors

17. Standard fixed point techniques are of no help in discharging these kinds of circularity, since what is at issue here is essentially self-*mention*, whereas although that terminology is commonly applied to recursive definitions, it would be more accurate to characterise recursion in terms of self-*use*.

genuine reflection, are not *causally connected* with the systems they model.

The reason that we cannot mix code for the underlying processor and code for the MCP and the reason that we ignored context arguments in the definitions above both have to do with the state of the processor P. In very simple systems (unordered rewrite rule systems, for example, and hardware architectures that put even the program counter into a memory location), the processor has no internal state, in the sense that it is in an identical configuration at every "click point" during the running of a program (i.e., all information is recorded explicitly in the structural field). But in more complex circumstances, there is always a certain amount of state to the processor that affects its behaviour with respect to any particular embedded fragment of code. In writing an MCP one must demonstrate, more or less explicitly, how the processor state affects the processing of object-level structures. By "more or less explicitly" I mean that the designer of the MCP has options: the state can be represented in explicit structures that are passed around as arguments within the processor, or it can be "absorbed" into the state of the processor running the MCP.[18]

The state of a processor for a recursively embedded functional language, of which Lisp is an example, is typically represented in an environment and a continuation, both in MCPs and in the standard metatheoretic accounts. (Note that these are notions that arise in the theory of Lisp, not in Lisp itself; except in self-referential or self-modelling dialects, user programs do not traffic in such entitles.) Most MCPs make the environment explicit. The control part of the state, however, encoded in a continuation, must also be made explicit in order to explain non-standard control operations, but in many MCPs (such as that in McCarthy (1965) and in Steele and Sussman's

18. I say that a property or feature of an object language is **absorbed** in a metalanguage or theory just in case the metatheory uses the very same property to explain or describe the property of the object language. Thus conjunction is absorbed in standard model theories of first-order logics, because the semantics of $P \wedge Q$ is explained simply by conjoining the explanation of P and Q—specifically, in such a formula as "'$P \wedge Q$' is true just in case 'P' is true *and* 'Q' is true".

«Add a note pointing to "The Correspondence Continuum"»

MCP for Scheme[19]) control context is absorbed.

Two versions of the 2-Lisp metacircular processor, one absorbing and one making explicit the continuation structure, are presented in sidebars on the following pages. Note that in both cases the underlying agency or anima is not reified; the "activity itself" remains entirely absorbed by the processor of the MCP. Nothing I have yet said (or in this paper will say) provides us with either name or mechanism to designate *process itself* (as opposed to structures and functional

19. See for example Sussman and Steele (1978b).

Non-Continuation-Passing 2-LISP Metacircular Processor

```
(define READ-NORMALISE-PRINT
     (lambda simple [env stream]
        (block (prompt&reply(normalise (prompt&read stream) env)
                             stream)
               (read-normalise-print env stream))))

(define NORMALISE
     (lambda simple [struc env]
        (cond  [(normal struc) struc]
               [(atom struc) (binding struc env)]
               [(rail struc) (normalise-rail struc env)]
               [(pair struc) (reduce (car struc) (cdr struc) env)])))

(define REDUCE
     (lambda simple [proc args env]
        (let [[proc! (normalise proc env)]]
           (selectq (procedure-type proc!)
              [simple (let [[args! (normalise args env)]]
                          (if (primitive proc!)
                              (reduce-primitive-simple proc! args!)
                              (expand-closure proc! args!)))]
              [intensional (if (primitive proc!)
                                (reduce-primitive-intensional proc!  args env)
                                (expand-closure proc!  args))]
              [macro (normalise  (expand-closure proc!  args) env))]))))

(define NORMALISE-RAIL
     (lambda simple [rail env]
        (if (empty rail)
            (rcons)
            (prep (normalise (1st rail) env)
                  (normalise-rail (rest rail) env)))))

(define EXPAND-CLOSURE
     (lambda simple [proc! args!]
        (normalise (body proc!)
                   (bind (pattern proc!) args! (environment proc!)))))
```

behaviour over structure), and no method of obtaining causal access to an independent locus of active agency has been (or will be) provided.[20]

---

20. The reason being that, as computer scientists, we as yet have no real theory of what processes are.
«Add a comment on this lack—and foreshadow work to come?»

Continuation-Passing 2-LISP Metacircular Processor

```
(define READ-NORMALISE-PRINT
    (lambda simple [env stream]
       (normalise (prompt&read stream) env
          (lambda simple [result]
             (block (prompt&reply result stream)
                (read-normalise-print env stream))))))

(define NORMALISE
    (lambda simple [struc env cant]
       (cond [(normal struc) (cont struc)]
          [(atom struc) (cont (binding struc env»]
          [(rail struc) (normalise-rail struc env cant)]
          [(pair struc) (reduce (car struc) (cdr struc) env cant)]])))

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

(define NORMALISE-RAIL
    (lambda simple [rail env cant]
       (if (empty rail)
           (cant (rcons))
           (normalise (1st rail) env
              (lambda simple [first!]
                 (normalise-rail (rest rail) env
                    (lambda simple [rest!]
                       (cant (prep first! rest!)))))))))

(define EXPAND-CLOSURE
    (lambda simple [proc! args! cant]
       (normalise(body proc!)
                (bind (pattern proc!) args! (environment proc!))
                cant)))
```

7 Procedural Reflection and 3·Lisp

Given the metacircular processors defined above, 3-Lisp can be non-effectively defined in a series of steps.

First, imagine a dialect of 2-Lisp, called 2-Lisp_1, where user programs are not run directly by the primitive processor, but by that processor running a copy of an MCP. Next, imagine 2-Lisp_2, in which the MCP in turn is not run by the primitive processor, but instead by the primitive processor running another copy of the MCP. And so on and so forth. 3-Lisp is essentially 2-Lisp_∞, except that the MCP is changed in a critical way in order to provide the proper connection between levels. 3-Lisp, in other words, is what I will call a **reflective tower**, defined as equivalent to an infinite number of copies of an MCP-like program, run at the "top" by an (infinitely fleet) processor. The claim that 3-Lisp is well-founded is the claim that the limit exists—that is, that both sides of the following equation are sound:

$$3\text{-Lisp} \approx \lim_{n \to \infty}\left(2\text{-Lisp}_\infty\right)$$

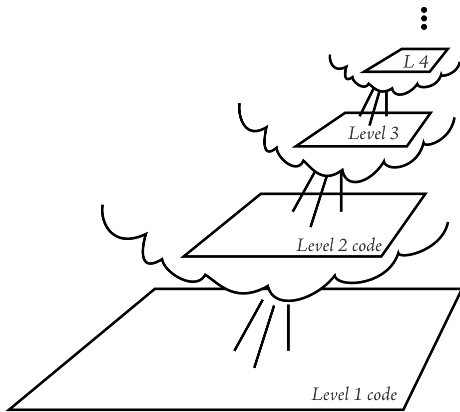

Figure 12 — The 3-Lisp Reflective Tower

I will explain the revised MCP presently, but first some general properties of this tower architecture. A rough idea of the levels of processing is given in figure 12: at each level the processor code is processed by an active process that interacts with it (locally and serially, as usual), but each processor is in turn composed of a structural field fragment in turn processed by a reflective processor on top of it. What I will show is that the implied infinite regress is not problematic, and that the architecture can be efficiently realised, since only a finite amount of information is encoded in all but a finite number of the bottom levels.

There are two ways to think about reflection. On the one hand, on what I will call the "**shifting view**," one can think of there being a primitive and noticeable **reflective act**, which causes the processor, in the sense of the basic locus of animating activity, to shift levels rather markedly either up or down, in what logicians and philoso-

phers might think of as *semantic ascent* and *semantic descent* (this is the explanation that best coheres with some of our pre-theoretic intuitions about reflective thinking, in the sense of contemplation). On the other hand, in what we might instead call the "**tower view**," which accords better with the explanation given in the previous paragraph, the model is instead of an infinite number of levels of reflective processors, each implementing the one below, without any shifting going on.[21] On this tower view, it is not coherent either to ask about *what level the tower is running at*, or to ask how many reflective levels are running: on the tower view *they are all running at once*. The same situation obtains when you use an editor implemented in APL. It is not as if the editor and the APL interpreter are both running together, either side-by-side or independently; rather, the one (the APL interpreter), being "interior" to the other, supplies the anima or agency of the outer one (the editor). To put this another way, when you implement one process in another process, you might want to say that you have two different processes, but you do not thereby have concurrency; the relation is is more like one of part and whole. It is just this sense in which the higher levels in our reflective hierarchy are always running: each of them is in some sense within the processor at the level below, so that it can thereby engender it.

I will not take a principled view on which account—a single locus of agency stepping between levels, or an infinite hierarchy of simultaneous processors—is correct, since they turn out to be behaviourally equivalent. Indeed, one way to characterise the model of reflection being proposed is as the following suggestion:

The semantically cleanest and most elegant way to understand a shifting reflective process is to model it as a tower. **[T]**

(One pragmatic rule of thumb: the simultaneous infinite tower of levels is often the better way to understand *processes*, whereas the shifting-level viewpoint is sometimes the better way to understand *programs*.)

21. Curiously, there are also intuitions about contemplative thinking, where one is both detached and yet directly present at the same time—which fit more closely with this view.

If we view 3-Lisp on the tower model, as an infinite reflective tower based on 2-Lisp, the code at each level can be understood as like the continuation-passing 2-Lisp MCP presented earlier,[22] but extended in an essential way: to provide a mechanism whereby the user's program can gain access to fully-articulated descriptions of that program's operations and structures. Thus extended, and appropriately located in a reflective tower, I will call this code the **3-Lisp reflective processor procedure (RPP)**. Programs gain reflective access to the articulated descriptions of the program's operations and structures by using what I will call **reflective procedures**—procedures that, when invoked, are: (i) run not at the level at which the invocation occurred, but one level higher, at the level of the reflective processor running the program; and (ii) given as arguments those structures being passed around in the reflective processor. I.e., reflective pro-

22. "Continuation-Passing 2-Lisp Metacircular Processor" sidebar, page ■■.

Programming in 3-Lisp

For illustration, we will look at a handful of simple 3-Lisp programs. The first merely calls the continuation with the numeral 3; thus a call to it (with no arguments) it is semantically identical to the simple numeral:

```
(define THREE
   (lambda reflect [[] env cant]
      (cant '3)))
```

Thus (THREE) ⇒ 3; (+ 11 (THREE)) ⇒ 14. The next example is an intensional predicate, true if and only if its argument (which must be a variable) is bound in the current context:

```
(define BOUND
   (lambda reflect [[var] env cont]
      (if (bound-in-env var env)
          (cont '$T)
          (cont '$F))))
```

or equivalently

```
(define BOUND
   (lambda reflect [[var] env cant]
      (cant ↑(bound-in-env var env))))
```

Thus (LET [[X 3]] (BOUND X)) ⇒ $T, whereas (BOUND X) ⇒ $F in the global context. The following quits the computation, by discarding the continuation and simply "returning":

cedures are essentially analogues of subroutines to be run "in the implementation," except that:

1. They are written in the same dialect as that being implemented;
2. They can use all the power of the implemented language in carrying out their function—i.e., reflective procedures can themselves make use of further reflective procedures, without limit;[23] and
3. Because they are within, not external to or "underneath" the architecture being implemented, they avoid all of the inelegance, implementation-dependence, and other deleterious

---

23. The tower is not a tower of different *languages*. There is a single dialect (3-Lisp) all the way up. What the tower is a tower of is *processors*—necessary because there is different processor state at each reflective level.

```
(define QUIT
   (lambda reflect [[□ env cant]
      'QUIT!))
```

There are a variety of ways to implement a `THROW/CATCH` pair; the following defines the version used in Scheme:

```
(define SCHEME-CATCH
   (lambda reflect [[tag body] catch-env catch-cant]
      (normalise
         body
         (bind tag
            ↑(lambda reflect [[answer] throw-env throw-cont]
               (normalise answer throw-env catch-cont))
            catch-env)
         catch-cant)))
```

For example:

```
(let [[x 1]]
   (+ 2 (scheme-catch punt
                  (* 3 (/ 4 (if (= x l)
                                (punt 15)
                                (- X 1)))))))
```

would designate seventeen and return the numeral 17.

The reflection mechanism is so powerful that many traditional primitives can be defined; `LAMBDA`, `IF`, and `QUOTE` are all non-primitive (user) definitions in 3-Lisp, defined as follows:

aspects of traditional code that has to "reach into the implementation" to do its work.

Reflective procedures are "defined" (in the sense described earlier) using the form

```
(LAMBDA REFLECT ARGS BODY)
```

where ARGS—typically the rail [ARGS ENV CONT]—is a pattern that should match a 3-element designator of, respectively, the argument structure at the point of call, the environment, and the continuation. Some simple examples are given in the "Programming in 3-Lisp" sidebar, above, including a fully functional definition of Scheme's CATCH. Though simple, these definitions would be impossible in a traditional language, since they make crucial access to the full processor state at point of call. Note also that although THROW and CATCH deal explicitly with continuations, the code that uses them need know nothing about such subtleties. More complex routines,

Programming in 3-Lisp (cont'd)

```
(define LAMBDA
   (lambda reflect [[kind pattern bodyJ env contJ
      (cont (ccons kind ↑env pattern body))))

(define IF
   (lambda reflect [[premise then elseJ env contJ
      (normalise premise env
         (lambda simple [premise!]
            (normalise (ef ↓premise! then else) env cant)))))

(define QUOTE
   (lambda reflect [[argJ env contJ (cant ↑arg)))
```

Some comments. First, the definition of LAMBDA just given is, of course, circular; a noncircular but effective version is given in Smith and des Rivières (1984); the one given above, if executed in 3-Lisp, would leave the definition unchanged, except that it is an innocent lie: in real 3-Lisp KIND is a procedure that is called with the arguments and environment, allowing the definition of (LAMBDA MACRO …), etc. CCONS is a closure constructor that uses SIMPLE and REFLECT to tag the closures for recognition by the reflective processor described in section 6. EF is an extensional conditional that normalises all of its arguments; the definition of IF defines the standard intensional version that normalises

such as utilities to abort or redefine calls already in process, are almost as simple. In addition, the reflection mechanism is so powerful that many traditional primitives can be defined, rather than having to be provided primitively: `LAMBDA`, `IF`, and `QUOTE` are all non-primitive (i.e., user) definitions in 3-Lisp, again illustrated in the sidebar. A simplistic break package is also presented, to illustrate the use of the reflective machinery for debugging purposes. It is noteworthy that *no* reflective procedures need be primitive; even `LAMBDA` can be built up from scratch.

The power and simplicity of these examples stems from the fact that the 3-Lisp reflective processor is causally connected in the right way, so as to allow the reflective procedures to run in the system in which they defined, rather than being models of another system. And, since reflective procedures are fully integrated into the system design (their names are not treated as special keywords), they can be passed around in the normal higher-order way. Finally, there is a

only one of the second two, depending on the result of normalising the first. And the definition of `QUOTE` will yield `(QUOTE A)` ⇒ `'A`.

Finally, we have a trivial break package, with `ENV` and `CONT` bound in the break environment for the user to see, and `RETURN` bound to a procedure that will normalise its argument and pass that out as the result of the call to `BREAK`:

```
(define BREAK
     (lambda reflect [[argJ env contJ
         (block (print arg primary-stream)
             (read-normalise-print "»"
                 (bind* ['env ↑envJ
                        ['cant ↑contJ
                        ['return ↑(lambda reflect [[a2J e2 c2]
                                      (normalise a2 e2 cont))]
                        env)
                 primary-stream))))
```

If viewed as models of control constructs in a language being implemented, these definitions will look innocuous; what is important to remember is that they work in the very language in which they are defined.

The 3-Lisp Reflective Processor Program (RPP)

```
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!
```

sense in which 3-Lisp is simpler than 2-Lisp, as well as being more powerful; there are fewer primitives, and 3-Lisp provides much more compact ways of dealing with a variety of intensional issues (like macros).

8 The 3- Lisp Reflective Processor

3-Lisp is best understood through a close inspection of the 3-Lisp reflective processor—the promised modification of the continuation-passing 2-Lisp metacircular processor mentioned above.

The code for the RPP is presented in a final sidebar, above. NORMALISE (line 7) takes a structure, environment, and continuation, and: (i)

returns the structure unchanged (i.e., sends it to the continuation) if it is in normal form; (ii) looks up the binding if it is an atom; (iii) normalises the structure's elements if it is a rail;[24] and (iv) otherwise reduces the `CAR` (procedure) with the `CDR` (arguments). `REDUCE` (line 13) first normalises the procedure, with a continuation (`C-PROC!`) that checks (line 17) to see whether it is reflective.[25] If it is not reflective, `C-PROC!` normalises the arguments, with a continuation that either expands the closure (lines 23–25) if the procedure is non-primitive, or else executes it directly (line 22) it if it is primitive.

As an example, consider `(REDUCE '+ '[X 3] ENV ID)`, assuming that `X` is bound to the numeral `2` and `+` to the primitive addition closure in `ENV`. At line 22, `PROC!` will designate the primitive addition closure, and `ARGS!` will designate the normal-form rail `[2 3]`. Since addition is primitive, we must simply do the addition. `(PROC! . ARGS!)` would not work, because `PROC!` and `ARGS!` are at the wrong level; they designate *structures*, not functions or arguments. For a brief instant, therefore, we dereference them (with ↓), do the addition, and then regain our meta-structural viewpoint with ↑.[26] If the procedure is

24. `NORMALISE-RAIL` is 3-Lisp's tail-recursive continuation-passing analogue of Lisp 1.5's `EVLIS`.

25. I adopt a convention of using exclamation point suffixes on atom names used as variables to designate normal form structures.

26. One way to understand this is to realize that the reflective processor simply asks its processor to do any primitives that it encounters—i.e., it passes responsibility for the execution of primitives up to the processor running it. In other words, each time one level uses a primitive, its processor runs around setting everything up, finally reaching the point at which it must simply *do* the primitive action, whereupon it asks its own processor for help. But, of course, that processor—i.e., the processor running the processor in question—will also come racing towards the edge of the same cliff, and will similarly duck responsibility, handing the primitive up yet another level.

The net result, from the "tower" perspective, is that every primitive ever executed is handed all the way to the (infinitely remote) top of the tower. There is then a magic moment, when the thing actually happens—and then the answer filters all the way back down to the level that started the whole procedure. It is as if the *deus ex machina*, living at the top of the tower, sends a lightning bolt down to some level or other, once every intervening level gets appropriately lined up (rather like the sun, at Stonehenge and the Pyramids, reaching down through a long tunnel at just one particular moment during the year).

reflective, however (line 18), it is called directly, not processed, and given the obvious three arguments (ARGS, ENV, and CONT) that are being passed around. ↓(DE-REFLECT PROC!) is merely a mechanism to "purify" the reflective procedure so that it does not reflect again, and to dereference it to be at the right level (we want to *use*, not *mention*, the procedure designated by PROC!). Note that line 18 is the only place that reflective procedures can ever be called; this is why they must always be prepared to accept exactly those three arguments.

This leads to an important point:

Reflective processor program line line 18 is the essence of 3-Lisp.

Line 18 alone engenders the full reflective tower, for it says that some parts of the object language—the code processed by this program—are called directly *in* this program. It is as if an object level fragment were included directly in the meta language, which raises the question of who is processing the meta language. This is where the tower enters the picture: the claim underlying 3-Lisp is that an *exactly equivalent reflective processor* is processing this code, too—and that this fact can be true without vicious threat of infinite ascent.

The result is to allow a reflective procedure "to be executed in the middle of the processor context." It is handed, as arguments, environment and continuation structures that designate the processing of the code below it, but it is *run* in a different context, with its own (implicit) environment and continuation, which are in turn represented in structures passed around by the processor one level above it. In this way a reflective procedure is given causal access to the state of the process that was in progress (answering one of the three initial requirements for reflection); as a result, it can cause any effect it wants since it has complete access to all future processing of that code. Furthermore, it has a safe place to stand, where it will not

---

Except, of course, that nothing ever happens, ultimately, except primitives. In other words the enabling agency, which must flow down from the top of the tower, consists of an infinitely dense series of these lightning bolts, with something like 10% of the ones that reach each level being allowed through that to the level below (and then 10% of those reaching to the level below it, etc.).

All infinitely fast.

«This should be edited to refer to the Implementation paper.»

conflict with the code being nm below it (thereby meeting the third criterion).

These various protocols illustrate a general point. As mentioned at the outset, part of designing an adequate reflective architecture involves a trade-off between being so connected that one steps all over oneself (as in traditional implementations of debugging utilities), and so disconnected (as with metacircular processors) that one has no effective access to what is going on. The suggestion made here is that the 3-Lisp reflective tower provides just the right balance between these two extremes, solving the problem of vantage point as well as of (both directions of) causal connection.

The 3-Lisp reflective processor unifies three traditionally independent capabilities in Lisp: (i) the explicit availability of EVAL and APPLY, (ii) the ability to support metacircular processors, and (iii) explicit operations provided for debugging purposes (such as MacLisp's RETFUN and Interlisp's FRETURN[27]). It is striking that the latter facilities are required in traditional dialects, in spite of the presence of the former, especially since they depend crucially on implementation details, violating portability and other natural aesthetics. In 3-Lisp, in contrast, all information about the state of the processor is fully available within the language itself—suggesting that its reflective architecture constitutes something of an appropriate theoretical unification of the kinds of extension that have heretofore had to be made in ad-hoc and non-transportable ways.

9 Threats of Infinity and Finite Implementations

The argument as to why 3-Lisp is finite is complex in detail, but simple in outline and substance. In brief: the proof relies on showing that the reflective processor is tail-recursive in two senses:

1. It runs programs tail-recursively, in that it does not build up records of state for programs across procedure calls (only on argument passing); and
2. It itself is fully tail-recursive, in the sense that all recursive calls within it (except for unimportant subroutines) occur in tail-recursive position.

27. «Refs?»

As a result, the reflective processor can be executed by a simple finite state machine. In particular—and this is the crucial point—it can run itself without using any state at all. Once the limiting behaviour of an infinite tower of copies of this processor has been determined, therefore,[28] that entire chain of processors can be simulated by another finite state machine, of complexity only moderately greater than that of the reflective processor itself.[29] A full copy of such an implementing processor[30] and a much more substantive discussion of tractability is provided in Smith & des Rivières (1984).

10 Conclusions and Morals

The use of Lisp as a language in which to explore programming semantics and reflection is not essential; the ideas should hold in any similar circumstance. I have chosen Lisp because it is familiar, because it has rudimentary self-referential capabilities, and because there is a standard procedural self-theory (continuation-passing metacircular "interpreters"). Work has begun, however, on designing reflective dialects of a side effect-free Lisp and of Prolog, and on studying a reflective version of the λ-calculus (the last being an obvious candidate to be used as a basis for a mathematical study of reflection).[k]

The techniques used here to define 3-Lisp can be generalised rather directly to these other languages. As suggested at the outset, in order to construct a reflective dialect one needs:

1. To formulate a theory of the language analogous to the metacircular processor descriptions we have examined;
2. To embed this theory within the language; and
3. To connect the theory with the underlying language in an appropriate causally connected way—i.e., so as to allow for

28. This has not yet been explained in this paper; see «refer to the implementation paper.»

29. It is an interesting open research question whether that "implementing" processor can be algorithmically derived from the reflective processor code.

 «Note that this has yet to be done … »

30. Consisting (including all utilities) of only about 200 lines of 2-Lisp code.

k) «May put in a sidebar on the result? I have it somewhere…»

> both "upwards" and "downwards" connection—by allowing reflective procedures invocable in the object language the ability to run (non-reflectively) in the processor (as was done in line 18 of the 3-Lisp reflective processor program).

It remains to implement the resulting infinite tower; a discussion of general techniques, which again would readily generalize to languages other than 3-Lisp, is presented in des Rivières and Smith (1984).

It is partly a consequence of using Lisp that I have used non-data-abstracted representations of functions and environments; this facilitates side effects to processor structures without introducing unfamiliar machinery. It is clear that environments could be readily abstracted, although it would remain open to decide what modifyonlylling operations would be supported (changing bindings is one, but one might wish to excise bindings completely, splice in new ones in, etc.). In standard l-calculus-based metatheory there are no side effects (and no notion of processing); environment designators must therefore be passed around ("threaded") in order to model environment side effects. It should be simple to define a side effect-free version of 3-Lisp with an environment-threading reflective processor, and then to define SETQ and other such routines as reflective procedures. Similarly, I have assumed in 3-Lisp a single structural field commonly visible from all code; one could define an alternative dialect in which the structural field, too, was threaded through the processor as an explicit argument, as in standard metatheory.

The representation of procedures as closures is troublesome.[31] I would be the first to admit that 3-Lisp provides too fine-grained (i.e., too metastructural) access to function designators—including continuations and the like. Given an appropriately abstract notion of procedure, it would be natural to define a reflective dialect that used abstract structures to encode procedures, and then to define reflective access in such terms. While I did not follow this direction here, in order to avoid taking on another very difficult problem, another intent of future work is to move in this direction.

31. Closures are failures, in a sense, in that they encode far more information than should be required in order to identify a function in intension; the problem being that we do not yet know what a function in intension might be.

These considerations all illustrate a general point: in designing a reflective processor, one can choose to bring into view more or less of the state of the underlying process. Fundamentally, it reduces to a design choice of what one wants to reify or make explicit, and what one wants to absorb. As currently defined, 3-Lisp reifies (i) the environment and (ii) the continuation, thereby making explicit those two implicit dimensions of processing one level below. It absorbs (iii) the structural field and (iv) the global environment; in addition, as mentioned earlier, it completely absorbs (v) the animating agency of the whole computation. If one were to define a reflective processor based on a metacircular processor that also absorbed the representation of control (in the style of the non-continuation-passing 2-Lisp MCP,[32] which embedded the control structure of the code being processed with the control structure of the processor), then reflective procedures would not have access to, and therefore could not affect, a base program's control structure. In any real application, it would need to be determined just what parts of the underlying dialect required reification.

More interestingly, one might be able to design a reflective language in which individual reflective procedures could specify, with respect to a very general meta-theory, which aspects they wanted explicit access to (simply environment in one case, animating agency in another, control structure but not agency in a third, etc.). In such a design, operations that needed only environmental access, such as BOUND?, would not need to traffic in continuations. While a modification of 3-Lisp that provides such "contextually optional" access to environment, continuation, and structural field, a full exploration of this possibility remains for future work.

One final point. Throughout this paper I have talked about semantics, but I have so far presented no mathematical semantical accounts of any of the dialect presented. To do so for 2-Lisp is relatively straightforward (see des Rivières and Smith (1984)[l]), but it remains to develop appropriate semantical equations to describe 3-Lisp. While might initially be tempting to construct such a model

32. Sidebar on p. ■■.

l) «Check; not sure this was ever done? Was it in the manual?»

based on the implementation strategy described in des Rivières and Smith (1984), I believe that doing so would be a failure. Instead, what is needed is a two-step process:

1. To construct a mathematical account of the "infinite tower" view of 3-Lisp—i.e., to take the limit as $n \to \infty$ of 2-Lisp$_n$, as suggested in §■■; and then
2. To prove, in terms of that model, that the finite implementation strategies presented in des Rivières and Smith (1984) are *correct*.

This awaits further work. Additional future work would include: (i) exploring what it would be to deal explicitly, in the semantical account, with anima or agency (rather than simply absorbing it), which would introduce parallelism into the reflective act; and (ii) formulating a more general account of the requisite causal connection, that are so crucial to the success of any reflective architecture. These various tasks will require more radical reformulations of semantics than have been considered here.

Acknowledgements

I have benefited greatly from the collaboration of Jim des Rivières on these questions, particularly with regard to issues of effective implementation. The research was conducted in the Cognitive and Instructional Sciences Group at the Xerox Palo Alto Research Center (PARC), as part of the Situated Language Program of Stanford's Center for the Study of Language and Information (CSLI).

References

Batali, John, "Computational Introspection," Artificial Intelligence Laboratory Memo AIM-TR-701, Massachusetts Institute of Technology, Cambridge, ma, 1983.

des Rivières, Jim and Smith, Brian Cantwell, "The Implementation of Procedurally Reflective Languages," 1984 Conference on LISP and Functional Programming, Austin, Texas, August 1984. Also available as Xerox Palo Alto Research Center (PARC) Report ISL–4, Palo Alto, CA (1984) and Stanford Center for the Study of Language and Information Report CSLI-84-9 (1984). Reprinted here as Chapter ■■.

Doyle, Jon, "A Model for Deliberation, Action, and Introspection," Artificial Intelligence Laboratory Memo AIM-TR-581, Massachusetts Institute of Technology, Cambridge, MA, 1980.

Fodor, Jerry. "Methodological Solipsism Considered as a Research Strategy in Cognitive Psychology," *The Behavioural and Brain Sciences*, 3:1 (1980) pp. 63–73; reprinted in Fodor, Jerry, *Representations*, Cambridge, MA: Bradford, 1981.

Genesereth, Michael and Lenat, Douglas B., "Self-Description and Modification in a Knowledge Representation Language," Heuristic Programming Project Report HPP-80-10, Stanford University Department of Computer Science, 1980.

McCarthy, John et al., *lisp 1.5 Programmer's Manual*. Cambridge, MA: MIT Press, 1965.

Smith, Brian Cantwell, *Reflection and Semantics in a Procedural Language*, Laboratory for Computer Science Report MIT-TR-272, 1982. Abstracts, Prologue, and Chapter 1 reprinted here as Chapter ■■.

Smith, Brian Cantwell and des Rivières, Jim, "Interim 3-Lisp Reference Manual," Report ISL-1, Xerox Palo Alto Research Center (PARC), Palo Alto, CA (1984…■■).

Steele, Guy, "LAMBDA: The Ultimate Declarative," Artificial Intelligence Laboratory Memo AIM-379, Massachusetts Institute of Technology, Cambridge, ma, 1976.

Steele, Guy and Sussman, Gerald, "The Revised Report on SCHEME, a Dialect of LISP," Artificial Intelligence Laboratory Memo AIM-452, Massachusetts Institute of Technology, Cambridge, ma, 1978a.

Steele, Guy and Sussman, Gerald, "The Art of the Interpreter, or, The Modularity Complex (parts Zero, One, and Two)," Artificial Intelligence Laboratory Memo AIM-453, Massachusetts Institute of Technology, Cambridge, ma, 1978b.

Weyhrauch, Richard W., "Prolegomena to a Theory of Mechanized Formal Reasoning," *Artificial Intelligence* 13:1,2 (1980) pp. 133–170.

2010 Perspective (cont'd)

Given the impossibility of bringing Mantiq to fruition, it was fortunate that 3-Lisp and procedural reflection were able to serve as the focus of a completable doctoral dissertation—though the advertising was disingenuous, since although Mantiq was genuinely supposed to be reflective, 3-Lisp ultimately amounted to being only what I would later call "introspective."[α6] (Mantiq was also intended to be descriptively as well as procedurally reflective; though I did recognize that 3-Lisp was limited to the procedural case.)

Some of the history of Mantiq and 3-Lisp is described in the Preface to the dissertation that resulted, published as a technical report under the name "Reflection and Semantics in Procedural Languages" (RSPL), *q.v.*[α7] Of special relevance here is the fact that the semantic orientation adopted in the 3-Lisp design, according to which programs are taken as *effective ingredients within computational processes*, rather than as external specifications of (or prescriptions for) them, was more familiar within knowledge representation (KR) and AI circles than it was in the programming language community per se. This perspective, which I dub an "ingrediential" view of programs, derives in part from the fact that I came to the Mantiq project out of an interest in knowledge representation, and that the KR community conceives its task as one of developing computer analogues of the mental structures that underlie active, real-world knowledge and thought processes—i.e., *as they occur during the course of a person's (or system's) ongoing life*—rather than as statically or once-and-for-all "specifying a mind," in the way that one might take to be the task of DNA. This ingrediential stance to reflection is quite explicit in RSPL, for example in the discussion of what I called the "Reflection Hypothesis":[α8]

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process can be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.*

At the time this 3-Lisp paper was published, I did not appreciate the theoretical significance, especially as regards semantics, of viewing programs from different perspectives. Recognition began to dawn soon thereafter, when I encountered the incomprehensibility with which my programming language colleagues greeted my approach to 2-Lisp (and thus 3-Lisp) semantics. A particularly telling event

occurred in 1984, when—proud of what I took to be its semantical cleanliness—I invited Joseph Goguen and Jose Meseguer, programming language theorists at SRI, to sketch out a "formal denotational semantics" for 2-Lisp. My plan was to use what they developed as a basis for initiating a mathematical analysis of 3-Lisp and reflection. When they generously came back with a proposal, however, I was—to be frank—astonished. What they took to be a mathematically clean semantical analysis obliterated what I took to be essential to 2-Lisp's semantical clarity—conflating distinctions I had taken such pains to maintain, such as among handles, numerals, and numbers, and between sequences and rails. Entities I took to be concrete were treated as abstract; the grounds on which I had rested my critique of the Lisp conception of evaluation had vanished; and in general their "theoretically clean" version of 2-Lisp had undergone a transformation that not only rendered it wholly unfamiliar to me, but that "disappeared" what was—at least in my eyes—its major contribution. Needless to say, , the proposed collaboration stalled, in spite of great respect on both sides (I mean nothing indicting by telling this tale; we were simply approach what we took to be a common subject matter from radically different perspectives). I never did develop a mathematical account of reflection—nor, to my knowledge, has anyone else.

Fortunately, in spite of this setback, the work on 3-Lisp and procedural reflection itself was kindly received in the larger community. After this paper appeared at the Principles of Programming Languages conference (POPL) in 1984, interest in reflection burgeoned around the world, and a variety of reflection conferences were held over the subsequent 10 years.[α9]

But the issues that had surfaced in the interaction with Goguen and Meseguer were a harbinger of more profound intellectual challenges than at the time I knew how to resolve. I had staked my dissertation on the fundamental thesis on which 3-Lisp is based (thesis [R], §1, p. (■■): *that reflection is relatively straightforward, if implemented on a semantically sound base.* While, in an overall sense, the topic of procedural reflection was widely picked up, that orienting thesis, with no exceptions of which I am aware, was resoundingly ignored.[α10] At first I was puzzled by people's blindness to or even dismissal of it,[α11] but I gradually came to appreciate that the incomprehensibility of this semantical thesis rested on the considerable conceptual difference of viewing programs as ingredients in computational processes, rather than as specifications or prescriptions of them.

As one would expect, the clearer I became on the underlying issues, the more I was able—especially in conversation—to explain the perspective from which

3-Lisp was designed. As I quickly learned, however, success in describing its architectures by and large required that I *not* use the ingrediential vocabulary I am employing here—i.e., depended on my not saying that the two dialects were based on a view of programs as causally effective process-internal ingredients. Rather, I had to describe them from a viewpoint that at the time felt alien to me: taking programs to be external, if nevertheless effective, process specifications or descriptions (or even prescriptions). A conversation with Gordon Plotkin (again in the mid 1980s) at Stanford's Center for the Study of Language and Information (CSLI) is illustrative. After failing to communicate anything about what mattered to me about 2-Lisp using my own terminology, I attempted to adopt his—i.e., tried to "inhabit" the specificational view—and said that what I was interested in was "*the semantics of the semantics of programs*." The ploy must have worked, as I recall him nodding and smiling. But the differences remained profound, and nothing further came of the conversation. Although I made some subsequent attempts to explain the differences in viewpoints (e.g., in (■■), it seems safe to say that the 2-Lisp and 3-Lisp approach to semantical clarity—and the idea of theorizing distinct procedural and declarative aspects of program meaning—was met with virtual silence when first presented, and then quickly faded into the background.

Over the intervening 25 years I have developed a much deeper understanding of these communicative failures, as well as an appreciation of the intellectual history that gave rise to them. The issues lie deep in the foundations of computing, and derive in part from the ways in which computer science has taken over technical terminology from philosophical and mathematical logic, but has used it for different purposes. Of numerous issues, one looms large in the present context: for reasons traceable as far back as Turing's original 1937 paper, computer scientists in general, and programming language theorists in particular, use what a classical logician would consider semantical vocabulary and model-theoretic techniques to analyse what that same logician would think of as fundamentally syntactic and/or proof-theoretic concerns. Disentangling this history helps to clarify all manner of communicative failures, theoretical confusions, and contextually incomprehensible behaviours—including such seemingly diverse topics as misunderstandings (on all sides) of Searle's Chinese Room thought experiment, the widespread use of constructive mathematics and intuitionistic logic in theoretical computer science (such as Martin-Löf's intuitionistic type theory) the structure of reflection, the meteoric rise in popularity (perhaps

even the provenance) of Girard's linear logic,[α12] and the substantial distraction we have all suffered, in my view, from focusing exclusively on the semantics of *programming languages*, rather than on the semantics of *individual programs*.

Elsewhere I have made some stabs at explaining these issues,[α13] but only briefly, and in passing. One of the goals of *The Age of Significance* (AOS) project,[α14] being launched as this is being written, is to spell out this history in ways that facilitate understanding across the boundaries of computer science—both "externally," as it were, by allowing what matters about computing to be understood from an external intellectual perspective, and "internally," by enabling the genuine semantical insights of the logical tradition to be appreciated within computer science (something that in my opinion has largely not yet occurred).

My exploration of these foundational issues has primarily taken place in my investigations into the philosophy of computing, and will be reported on as such. More technically, after the publication of this paper my attention did not stay focused on programming languages, but turned back towards the issues that had originally motivated Mantiq: how to generalize the lessons learned here in the context of people and/or systems able to reason about the concrete, external world.

I was sobered not only by the daunting challenges of doing justice to real-world metaphysics and ontology, but also by an inadvertent lesson gained from the 3-Lisp exercise: the untenable pedantry of excessive semantical strictness. Not only was it manifest that dealing with real-world ontology was a profoundly more serious challenge than anything for which the 3-Lisp project provided preparation, but it also quickly became clear that semantics itself, *and any ideal of "semantical clarity,"* would have to be rethought in the most fundamental way, if we were even to approach, in artificial systems, the prowess and facility with which we people think about and find intelligible the worlds in which we are embedded. Some initial steps in these directions were reported in "The Correspondence Continuum" and "Varieties of Self-Reference," both written in 1986.[α15] But as noted in the annotations to those papers included in this volume, I ultimately came up against what I came to call an "ontological wall,"[α16] prompting me to delve even deeper into epistemology and metaphysics—a shift in emphasis that led to the writing of *On the Origin of Objects* (O3) in 1996,[α17] and that continues to this day.

I do not believe it would be impossible to incorporate at least some of the lessons of O3 in a reflective computational system—in part because of not believing

that 'computational' is a restrictive property (see AOS). But until such a day—a day that it is hard to know whether I myself will ever reach—the original motivations for developing 3-Lisp, the fundamental insights on which it is based, and the original vision of Mantiq all remain waiting in the wings.

Notes

α1 Sidebars and footnotes with text in sans-serif font, as in this case, contain comments and reflections added in 2010, rather than material that appeared in the original paper.]

α2 'Mantiq' (منطق) is roughly the Arabic equivalent of the Greek *logos* (λογοσ)—meaning *speech*, *manner of speaking*, *eloquence*, or *logic* «ref: *The Hans Wehr Dictionary of Modern Written Arabic*). It is best known in the title *Mantiq al-Tayr* (منطق الطير), a book of poems by the Sufi poet Farid al-Din Attar, sometimes translated as "The Language of the Birds" but more commonly as "The *Conference* of the Birds."

α3 At least what philosophers would call its "narrow" meaning (cf. «ref»). Not only did I quickly come to realise that a great variety of different things been called the "meaning" of an expression or idea, over the years, but I have also come to believe there never will be a "final catalogue" of just which of the infinite number of aspects of an intentional utterance or event can or do matter to its full significance. Even more challenging, from a design point of view, I believe that what we take to be the "meaning" of such any such event or occassion (let alone what "type" it instantiates) is likely contextually dependent not only on facts about the event so taken, but on the circumstances of the situation in which the meaning is referred to.

Moreover, whatever eventual story about meaning one were to adopt, it is likely that a true "fusion" of meaning and structural identity would prove impossible in the limit, since it is usually possible, given any such view, to construct examples showing that meaning identity is uncomputable. Still, having some such goal as an ideal can provide motivation and direction towards "higher-level" archichtures of intentional capacity.

α4 The first drafts of the report on 3-Lisp were designed to be chapter 13 of the infeasible Mantiq dissertation.

α5 The idea can clearly be generalised, allowing one to "step sideways," as it were, so as to be able to see one whole tower as a unity, etc. But I say "first good idea" because I was interested in a much more radical kind of reflection, involving a wholesale "leap" across a chasm from one locus of intelligibility to another, which (by definition) cannot be "viewed" from a vantage point accessible within the "prior" epistemic architecture. The merest sketch of such an idea is mentioned in O3 «ref»; I plan to explore it much more fully in Phase II of AOS «ref».

α6 See "Varieties of Self-Reference," Chapter ■■.

α7 Reprinted here as chapter ■■. The dissertation itself was submitted as "Procedural Reflection in Programming Languages'; the change in title reflected not only the importance of thesis [R] (p. ((), but also my increasing awareness of the importance of the semantical model on which the reflective architecture was based.

α8 Op. cit, pp. ■■.

α9 «References»

α10 For example, although the Wikipedia web page on reflection in computer science (below) credits the 3-Lisp work as introducing the notion of reflection into programming languages, it makes no mention of the rationalised semantics on which the 3-Lisp design was based (in spite of discussion throughout the article about the "subject matter" of programming constructs). Similarly, none of the ten examples of reflection in contemporary languages presented at the end of the article are designed in terms of an explicit theorization of subject matter or declarative import.

http://en.wikipedia.org/wiki/Reflection_(computer_science)

α11 Cf. Daniel P. Friedman and Mitchell Wand, "Reification: Reflection without Metaphysics," LISP and Functional Programming Conference, 1984, pp 348-55.

α12 «References»

α13 E.g., in "The Foundations of Computing," reprinted here as chapter ■■.

α14 See http://www.ageofsignificance.org

α15 See chapter ■■ and chapter ■■.

α16 E.g., see "The Foundations of Computing," reprinted here as chapter ■■.

α17 On the Origin of Objects, MIT Press, Cambridge, MA: 1996.