Reflection and Semantics in Lisp

Brian Cantwell Smith

XEROX Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304; and
Center for the Study of Language and Information
Stanford University, Stanford, CA 94305

1. Introduction

For three reasons, Lisp's self-referential properties 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. 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 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 embeddings of Lisp in Lisp 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 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 [Steele and Sussman 1978b], except connected to each other in a simple but critical way. In such an architecture, any aspect of a process's state that can be described in terms of the theory can be rendered explicit, in program accessible structures. Furthermore, as we will see, this apparently infinite architecture can be finitely implemented.

The architecture allows the user to define complex programming constructs (such as escape operators, deviant variable-passing protocols, and debugging primitives), by writing direct analogues of those 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 this case I have used a theory of Lisp based on environments and continuations. A 3-Lisp program, therefore, at any point during a computation, can obtain representations of the environment and continuation characterising the state of the computation at that point. Thus, such constructs as THROW and CATCH, which must otherwise be provided primitively, can in 3-Lisp be easily defined 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 them). And all this can be done without writing 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 Steele's example, continuations must be passed around explicitly from the beginning). Rather, 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. It provides a mechanism, in other words, of reaching up and "pulling information out of the sky" when unexpected circumstances warrant it, without having to worry about it otherwise.

The overall claim is that reflection is simple to build on a semantically sound base, where 'semantically sound' means more than that the semantics be carefully formulated. Rather, I assume throughout that computational structures have a semantic significance that transcends their behavioural import -- or, to put this another way, that computational structures are about something, over and above the effects they have on the systems they inhabit. Lisp's NIL, for example, not only evaluates to itself forever, but also (and somewhat independently) stands for Falsehood. A reconstruction of Lisp semantics, therefore, must deal explicitly with both declarative and procedural aspects of the overall significance of computational structures. This distinction is different from (though I will contrast it with) the distinction between operational and denotational semantics. It is a reconstruction has been developed within a view that programming languages are properly to be understood in the same theoretical terms used to analyse not only other computer languages, but even natural languages.

This approach forces us to distinguish between a structure's value and what it returns, and to discriminate entities, like numerals and numbers, that are isomorphic but not identical (both instances of the general intellectual hygiene of avoiding use/mention errors). Lisp's basic notion of evaluation, I will argue, is confused in this regard, and should be replaced with independent notions of designation and simplification. The result is illustrated in a *semantically rationalised* dialect, called 2-Lisp, based on a simplifying (designation-preserving) term-reducing processor. The point of defining 2-Lisp is that the reflective 3-Lisp can be very simply defined on top of it, whereas defining a reflective version of a non-rationalised dialect would be more complicated and more difficult to understand.

The strategy of presenting a general architecture 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, we 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 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 particular, with recursion — a construct with which it shares many features. Specifically, recursion can seem viciously circular to the uninitiated, and can lead to confused implementations if poorly understood. The mathematical theory of recursion, however, 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, initially seem viciously circular (or at least infinite), and are difficult to implement without an adequate understanding. The intent of this paper, however, is to argue that reflection is 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 lay out some motivations and assumptions. First, 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; 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[1] that reflective abilities underlie much of the plasticity with which we deal with the world, both in language (such as when one says *Did you understand what I meant?*) and in thought (such as when one wonders how to deliver bad news compassionately). Common sense suggests that reflection enables us to master new skills, cope with incomplete knowledge, define terms, examine assumptions, review and distill our experiences, learn from unexpected situations, plan, check for consistency, and recover from mistakes.

In spite of working with reflection in formal languages, most of the driving intuitions about reflection are grounded in human rationality and language. Steps towards reflection, however, can also be found in much of current computational practice. Debugging systems, trace packages, dynamic code optimizers, run-time compilers, macros, metacircular interpreters, error handlers, type declarations, escape operators, comments, and a variety of other programming constructs involve, in one way or another, 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 make claims. It is the hope that the lessons learned in this smaller task will serve well in the larger account.

We mentioned at the outset that the general task, in defining a reflective system, is to embed a theory of the system in the system, so as to support smooth shifting between reasoning directly about the world and reasoning about that reasoning. Because we are talking of reasoning, not merely of language, we added an additional requirement on this embedded theory, beyond its being descriptive and true: it must also be what we will call *causally connected*, so that accounts of objects and events are tied directly to those objects and events. The

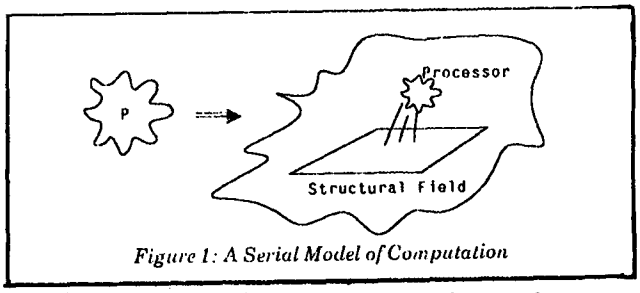


Figure 1: A Serial Model of Computation

causal relationship, furthermore, must go both ways: from event to description, and from description back to event. (It is as if we were creating a magic kingdom, where from a cake you could automatically get a recipe, and from a recipe you could automatically get a cake.) In mathematical cases of self-reference, including both self-referential statements, and models of syntax and proof theory, there is of course no causation at all, since there is no temporality or behaviour (mathematical systems don't run). Causation, however, is certainly part of any reflective agent. Suppose, for example, that you capsize while canoeing through difficult rapids, and swim to the shore to figure out what you did wrong. You need a description of what you were doing at the moment the mishap occurred; merely having a name for yourself, or even a general description of yourself, would be useless. Also, your thinking must be able to have some effect; no good will come from your merely contemplating a wonderful theory of an improved you. As well as stepping back and being able to think about your behaviour, in other words, you must also be able to take a revised theory and "dive back in under it", adjusting your behaviour so as to satisfy the new account. And finally, we mentioned that when you take the step backwards, to reflect, you need a place to stand with just the right combination of connection and detachment.

Computational reflective systems, similarly, must provide both directions of causal connection, and an appropriate vantage point. Consider, for example, 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. First, stack-frames and implementation codes are really 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, they are always available, and always true. They have these properties because they play a causal role in the very existence of the process they implement; they therefore automatically solve the "event-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. Without this 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 variable clashes and other unwanted interactions.

As we will see in developing 3-Lisp, all of these concerns can be dealt with in a reflective language in ways that are both simple 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

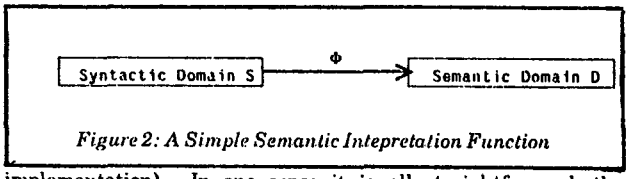


Figure 2: A Simple Semantic Intepretation Function

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 is evident, but with how such power can be finitely provided — a question we will examine 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 collectively called 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, I claim here only that our particular architecture 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 inside the machine, never for abstract mathematical or other "external" entities like numbers, functions, or radios. (Although this terminology may be confusing for semanticists who think of a structure as a model, I want to avoid calling them *expressions*, since the latter term connotes linguistic or notational entities. The aim is for a concept covering both data structures and internal representations of programs, with which to categorize what we would in ordinary English call the *structure* of the overall process or agent.) Consequently, I call *metastructural* any structure that designates another structure, reserving *metasyntactic* for expressions designating linguistic entities or expressions.[2] Given our interest in internal self-reference, it is clear that both structural field and processor, as well as numbers and functions and the like, will be part of the semantic domain. Note that metastructural calculi must be distinguished from those that are higher-order, in which terms and arguments may designate functions of any degree (2-Lisp and 3-Lisp will have both properties).[3]

3. A Framework for Computational Semantics

We turn, then, 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 called an *interpretation function* (to be distinguished, as noted above, from the standard computer science notion of an *interpreter*). It is usually specified inductively, with respect to the compositional structure of the elements of the syntactic domain, which is typically a set of syntactic or linguistic sorts of entities. The semantic domain may be of any type whatsoever, including a domain of behaviour; in reflective systems it will often include the syntactic domain as a proper part. We will use a variety of different terms for different kinds of semantic relationship; in the general case, we will call **s** a *symbol* or *sign*, and say that **s** *signifies* **d**, or conversely that **d** is the *significance* or *interpretation* of **s**.

In a computational setting, there are several semantic relationships — not different ways of characterizing 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, as do ambiguities in the use of words like 'program'. In many settings, such as in purely extensional functional programming languages, such distinctions are inconsequential. But when we turn to reflection, self-reference, and metastructural processors, these otherwise minor distinctions play a much more important role. Also, since the semantical theory we adopt will be at least partially embedded within 3-Lisp, the analysis will affect the formal design. Our approach, therefore, will be start with basic and simple intuitions, and to identify a finer-grained set of distinctions than are usually employed. We will consider very briefly the issue of how current programming language semantics would be reconstructed in these terms, but the complexities involved in answering that question adequately would take us beyond the scope of the present paper.

At the outset, we distinguish three things: a) the objects and events in the world in which a computational process is embedded, including both real-world objects like cars and caviar, and set-theoretic abstractions like numbers and functions (i.e., we adopt a kind of pan-platonic idealism about mathematics); b) 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); and c) 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 last set are the consitituents of the communication one has with the computational process; the middle are the ingredients of the process with which one interacts, and the first (at least presumptively) are the elements of the world about which that communication is held. In the human case, the three domains correspond to world, mind, and language.

It is a truism that the third domain of objects — communication elements — are semantic. We claim, however, that the middle set are semantic as well (i.e., that structures are bearers of meaning, information, or whatever). Distinguishing between the semantics of communicative expressions and the semantics of internal structures will be one of main features of the framework we adopt. It should be noted, however, that in spite of our endorsing the reality of internal structures, and the reality of the embedding world, it is nonetheless true that the only things that actually happen with computers (at least the only thing we will consider, since we will ignore sensors and manipulators) are communicative interactions. If, for example, I ask my Lisp machine to calculate the square root of 2. what I do is to type some expression like `(SQRT 2.0)` at it, and then receive back some other expression, probably quite like `1.414`, by way of response. The interaction is carried out entirely in terms of expressions; no structures, numbers, or functions are part of the interactional event. The participation or relevance of any of these more abstract objects, therefore, must be inferred from, and mediated through, the communicative act.

We will begin to analyse this complex of relationships using the terminology suggested in Figure 3. By $\mathcal{O}$, very simply, we refer to the relationship between external notational expressions and internal structures; by Ψ to the processes and behaviours those structural field elements engender (thus Ψ is inherently temporal), and by Φ to the entities in the world that they designate. The relations Φ and Ψ are named, for mnemonic convenience, by analogy with 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, in contrast, are "within the head" (of person or machine).

Computation is inherently temporal; our semantic analysis, therefore, will have to deal explicitly with relationships across the passage of time. In Figure 4, therefore, we have unfolded the diagram of Figure 3 across a unit of time, so as to get at a full configuration of these relationships. The expressions $\mathbf{n}_1$ and $\mathbf{n}_2$ are intended to be linguistic or communicative entities, as described above; $\mathbf{s}_1$ and $\mathbf{s}_2$ are internal structures over which the internal processing is defined. The relationship $\mathcal{O}$, which we will call *internalisation*, relates these two kinds of object, as appropriate for the device or process in question (we will say, in addition, that $\mathbf{n}_1$ *notates* $\mathbf{s}_1$). For example, in first-order logic $\mathbf{n}_1$ and $\mathbf{n}_2$ would be expressions, perhaps written with letters and spaces and '$\exists$' signs; $\mathbf{s}_1$ and $\mathbf{s}_2$, to the extent they can even be said to exist, would be something like abstract derivation tree

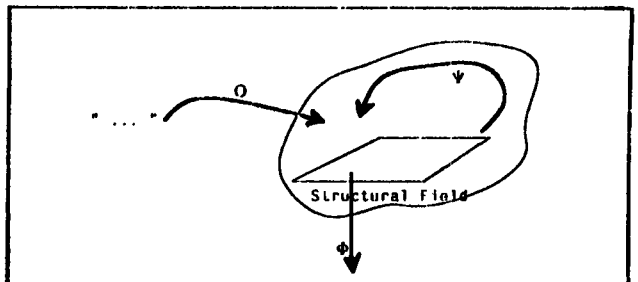


Figure 3: Semantic Relationships in a Computational Process

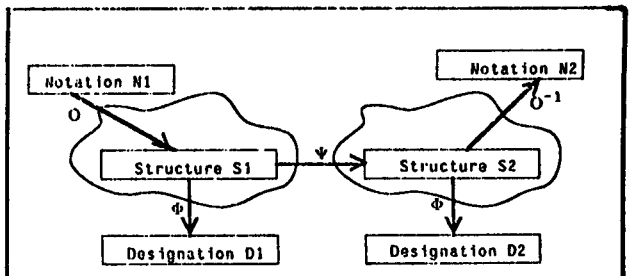


Figure 4: A Framework for Computational Semantics

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 cons-cells in the s-expression heap.

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 the interpretation function that makes explicit what we will call the *designation* of internal structures (not the designation of linguistic terms, which would be described by $\Phi \circ O$). The relationship between my mental token for 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 $\Phi(O(\text{"T.S.ELIOT"})) = \text{T.S.ELIOT}$. Similarly, Φ would relate an internal "numeral" structure (say, the numeral 3) to the corresponding number. As mentioned at the outset, our focus on Φ is evidence of our permeating semantical assumption that all structures have designations — or, to put it another way, that the structures are all symbols.[4]

The Ψ relation, in contrast to O and Φ, always (and necessarily, because it dosen't have access to anything else) relates some internal structures to others, or at least to behaviours over them. To the extent that it would make sense to talk of a Ψ in logic, it would approximately be the formally computed derivability relationship (i.e., $\vdash$); 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., Ψ is evaluation in Lisp).

The relationships O, Ψ, and Φ have different relative importances in different linguistic disciplines, and different relationships among them have been given different names. For example, O 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 (which, incidentally, can be expressed as the equation $\Psi(s_1,s_2) \equiv [\, d_1 \subseteq d_2 \,]$, if one takes Ψ to be a relation, and Φ to be an inverse satisfaction relationship between sentences and possible worlds that satisfy them). 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 [Fodor 1980]. That O is usually specifiable compositionally and independently of Φ or Ψ is essentially a statement of the autonomy thesis for language. Similarly, when O cannot be specified indepently of Ψ, computer science will say that a programming language "cannot be parsed except at runtime" (Teco and the first versions of Smalltalk were of this character).

A thorough analysis of these semantic relationships, however, and of the relationships among them, is the subject of a different paper. For present purposes we need not take a stand on which of O, Ψ, or Φ has a prior claim on being semantics, but we do need a little terminology to make sense of it all. For discussion, we will refer to the "Φ" of a structure as its *declarative import*, and to its "Ψ" as its *procedural consequence*. 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, we will say that s_1 is *self-referential* if $d_1 = s_1$, 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 Ψ — α- and β-reduction — do not alter the interpretation in the standard model).

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 will be considered in 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. 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 communicative object, not so much playing a role within a computational process as existing outside the process and representing it. Putting aside for a moment the question of whom it is meant to communicate to, we would simply say that a program is in the domain of Θ, and, roughly, that $\Phi \circ \Theta$ 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 4), since specifications aren't themselves 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 $\Phi(\Theta(b)) = c$. 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, however, an alternative conception, that places the program inside the machine as a causal participant in the behaviour that results. This view is closer to the one implicitly adopted in Figure 1, and it is closer (we claim) to the way in which a Lisp program must be semantically analysed, especially if we are to understand Lisp's emergent reflective properties. In some ways this different view has a von Neuman character, in the sense of equating program and data. On this view, the more appropriate equation would seem to be $\Psi(O(b)) = c$, 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; something it is not clear it is possible to do.

But this will require further work. What we can say here is that programming language semantics seems to focus on what, in our terminology, would be an amalgam of Ψ and Φ. For our purposes we need only note that we will have to keep Ψ and Φ strictly separate, while recognising (because of the context relativity and nonlocal effects) that the two parts cannot be told independently. Formally, one needs to specify a *general significance function* Σ, that recursively specifies Ψ and Φ 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 relationship to the world that s_1 signifies. 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 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 numeral 2.

Before leaving semantics completely, it is instructive to apply our various distinctions to traditional Lisp. We said above that all interaction with computational processes is mediated by communication; this can be stated in this terminology by noting that O and O^{-1} (we will call the latter *externalisation*) are a part of any interaction. Thus Lisp's "read-eval-print" loop is mirrored in our analysis as an iterated version of $O^{-1} \circ \Psi \circ O$ (i.e., if n_1 is an expression you type at Lisp, then n_2 is $O^{-1}(\Psi(O(n_1)))$). 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) object, with its own syntax (of parentheses and identifiers in the lexical case; of boxes and arrows in the graphical).

There is in Lisp 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 change the CAR of a field structure; it will not back up your terminal and erase the eleventh character of your input expression. Similarly, Lisp atoms are field elements, not to be confused with their lexical representations (called P-names). Again, quoted forms like (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 ("ABC").[5]

4. Evaluation Considered Harmful

The claim that all three relationships (O, 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. Nonetheless, it is unarguable that Lisp's numerals designate numbers, and that the atoms T and NIL (at least in predicative contexts) designate truth and falsity — no one could learn Lisp without learning this fact. Similarly, (EQ 'A 'B) designates falsity. Furthermore, 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's a procedure", where the term "the CAR of the list" is used as an English referring expression, 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 designating the atom A; that's just what quotation is. Finally, we can take atoms like CAR and + to designate the obvious functions.

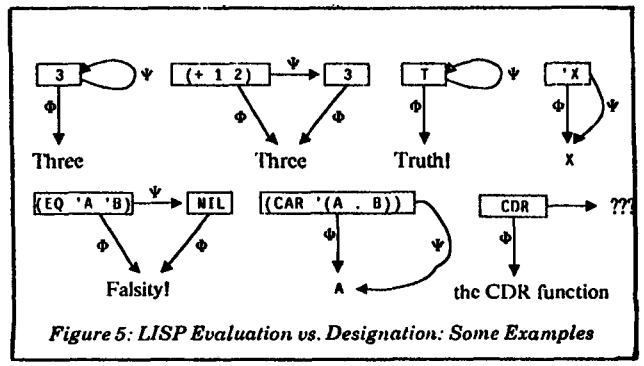


Figure 5: LISP Evaluation vs. Designation: Some Examples

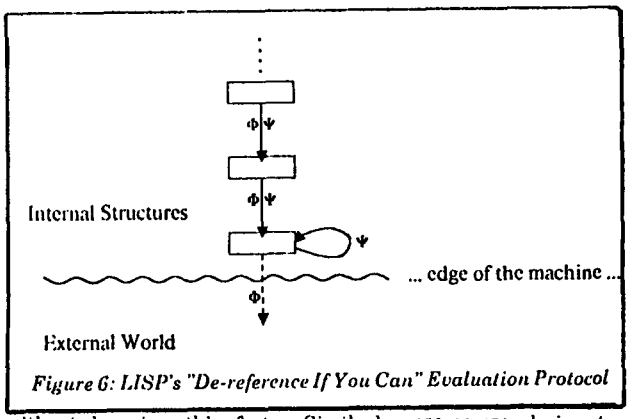


Figure 6: LISP's "De-reference If You Can" Evaluation Protocol

What, then, is the relationship between the declarative import (Φ) of Lisp structures and their procedural consequence (Ψ)? Inspection of the data given in Figure 5 shows that Lisp obeys the following constraint (more must be said about Ψ in those cases for which $\Phi(\Psi(s)) = \Phi(s)$, since the identity function would satisfy this equation):

$$\forall s \in S\,[\,\text{if}\,[\Phi(s) \in S]\ \text{then}\,[\Psi(s) = \Phi(s)]\ \text{else}\,[\Phi(\Psi(s)) = \Phi(s)]\,] \qquad (1)$$

All Lisps, including Scheme [Steele and Sussman 1978a], in other words, dereference any structure whose designation is another structure, but will return a co-designating structure for any whose designation is outside of the machine (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***, evaluation in Lisp is often a designation-preserving operation. In fact no computer can evaluate a structure like (+ 2 3), if that means returning the designation, any more than it can evaluate the name *Hesperus* or *peanut butter*.

Obeying equation (1) is highly anomolous. It means that even if one knows what Y is, and knows X evaluates to Y, one still doesn't know what X designates. It licences such semantic anomalies as (+ 1 '2), which will evaluate to 3 in all extant Lisps. Informally, we 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 the co-domain is limited to the structural field), it seems they should always simplify, and therefore obey the following constraint (diagrammed in Figure 7):

$$\forall s \in S\,[\Phi(\Psi(s)) = \Phi(s) \wedge \text{NORMAL-FORM}(\Psi(s))] \qquad (2)$$

The content of this equation clearly depends entirely on the content of the predicate NORMAL-FORM (if NORMAL-FORM were λX.true then Ψ could be the identity function). In the λ-calculus, the

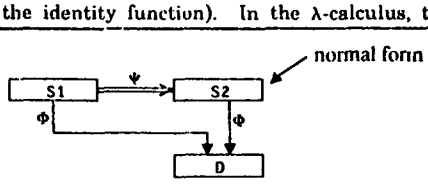


Figure 7: A Normalisation Protocol

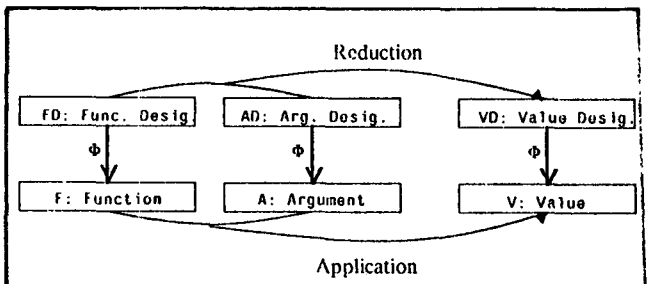


Figure 8: Application vs. Reduction

notion of normal-formedness is defined in terms of the processing protocols (α- and β-reduction), but we cannot use that definition here, on threat of circularity. Instead, we 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 normalise 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.

Two notes. First, I won't 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 a function from structures and arguments onto values, but its use, like 'evaluate', is rife with use/mention confusions. As illustrated in Figure 8, we will use 'apply' for mathematical function application — i.e., to refer to a relationship between a function, some arguments, and the value of the function applied to those arguments — and the term 'reduce' to relate the three expressions that designate functions, arguments, and values, respectively. Note that I still use the term 'value' (as for example in the previous sentence), but only to name that entity onto which a 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. 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 is merely what it is. Given one account of what it does (like an implementation), one can compare that to another account (like a specification). One can also prove that it has certain properties, such as that it always terminates, or uses resources in certain ways. One can 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 fundamental question about the semantical nature of the processor itself. We are not looking for a way in which to say that the semantics of (CAR '(A . B)) is A because that is how the language is defined; rather, we want to say that the language was defined that way because A is what (CAR '(A . B)) designates. Semantics, in other words, can be a tool with which to judge systems, not merely a method of describing them.

5. 2-Lisp: A Semantically Rationalised Dialect

Since we have torn apart the notion of evaluation into two constituent notions, we must start at the beginning and build Lisp over again. 2-Lisp is a proposed result. Some summary comments can be made. 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 called *category alignment*). More specifically, Lisp expressions are sorted into categories by notation, by structure (atoms, cons pairs, numerals), by procedural treatment (the "dispatch" inside EVAL), and by declarative semantics (the type of object designated). Traditionally, as illustrated in Figure 9, these categories are not aligned; lists, a derived structure type, include some of the pairs and one atom (NIL); the procedural regimen 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 we require the notational, structural, procedural, and semantic categories to correspond one-to-one, as shown 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).

A summary of 2-Lisp is given in Figure 11, but some comments can be made here. Like most mathematical and logical languages, 2-Lisp is almost entirely declaratively extensional. Thus (+ 1 2), which is 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 one can think of as "procedurally extensional"), in the sense that procedures by and large normalise their arguments. Thus, (+ 1 (BLOCK (PRINT "hello") 2) will normalise to 3, printing 'hello' in the process.

Many properties of Lisp that must normally be posited in an ad hoc way fall out directly from our analysis. For example, one must normally state explicitly that some atoms, such as T and NIL and the 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 the 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 five and normalise to 5; no metatheoretic machinery is needed for this "uncurrying" operation (in regular Lisp one must use (APPLY '+ X); in Scheme, (APPLY + X)).

There are numerous properties of 2-Lisp that we will ignore in this paper. The dialect is defined (in [Smith 82]) to include side-effects, intensional procedures (that do not normalise their arguments), and a variety of other sometimes-shunned properties, in part to show that our semantic reconstruction is compatible with the full gamut of features found in real programming languages. Recursion is handled with 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

We 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,

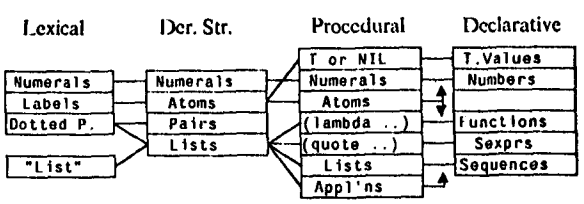


Figure 9: The Category Structure of LISP 1.5

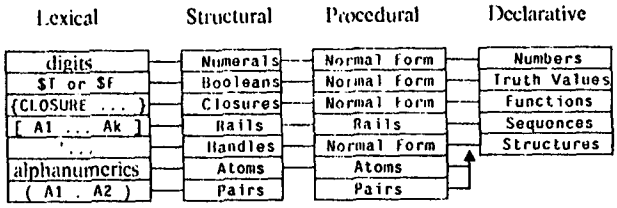


Figure 10: The Category Structure of 2-LISP and 3-LISP

Figure 11: An Overview of 2-Lisp

We begin with the 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 we distinguish them from pairs in order to avoid category confusion, and give them their own name, in order to avoid confusion with sequences (or vectors or tuples), 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 it may be). *Pairs* (sometimes also called *redexes*, and notated '(A_1 . A_2)') designate the value of the function designated by the CAR applied to the arguments designated by the CDR. By taking the notational form '(A_1 A_2 ... A_k)' to abbreviate '(A_1 . [A_2 A_3 ... A_k])' instead of '(A_1 . (A_2 . (... (A_k . NIL)...)))', we preserve the standard look of Lisp programs, without sacrificing category alignment. (Note that in 2-Lisp there is no distinguished atom NIL, and '() is a *notational* error — corresponding to no structural field element.) *Closures* (notated '{CLOSURE: ... }') are normal-form function designators, but they are not canonical, since it is not generally 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', ''(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 a QUOTE procedure is easy to define in 3-Lisp).

We 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 unmotivated 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 3] [1 2 3]) ⇒ $T
(= '[1 2 3] '[1 2 3]) ⇒ $F
(= [1 2 3] '[1 2 3]) ⇒ $F

(In the last case one structure designates a sequence and one a rail.) 1ST and REST are the CAR/CDR analogues on sequences and rails; thus, (1ST [10 20 30]) ⇒ 10; (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 called ACONS, RCONS, and CCONS. There are 11 primitive characteristic predicates, 7 for the internal structural types (AIOM, PAIR, RAIL, BOOLEAN, NUMERAL, CLOSURE, and HANDLE) and 4 for the external types (NUMBER, TRUTH-VALUE, SEQUENCE, and FUNCTION). Thus:

(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 keyword SIMPLE will be explained presently); thus (LAMBDA SIMPLE [X] (+ X X)) returns 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. However, as mentioned earlier 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 notated with the arrows '↑' and '↓') help to mediate this metastructural hierarchy (there is otherwise no way to add or remove quotes; '2 will normalise to '2 forever, never to 2). Specifically, ↑*STRUC* designates the normal-form designator of the designation of *STRUC*; i.e., ↑*STRUC* designates what *STRUC* normalises to (therefore ↑(+ 2 3) ⇒ '5). Thus:

(LAMBDA SIMPLE [X] X) designates a function,
'(LAMBDA SIMPLE [X] X) designates a pair or redex, and
↑(LAMBDA SIMPLE [X] X) designates a closure.

(Note that '↑' is call-by-value but not declaratively extensional.) Similarly, ↓*STRUC* designates the designation of the designation of *STRUC*, providing the designation of *STRUC* is in normal-form (therefore ↓'2 ⇒ 2). ↓↑*STRUC* is always equivalent to *STRUC*, in terms of both designation and result; so is ↑↓*STRUC* when it is defined. Thus if DOUBLE is bound to (the result of normalising) (LAMBDA [X] (+ X X)), then (BODY DOUBLE) generates an error, since BODY is extensional and DOUBLE designates a function, but (BODY ↑DOUBLE) will designate the pair (+ X X).

Type	*Designation*	*Normal*	*Canonical*		*Notation*
Numerals	Numbers	Yes	Yes	—	*digits*
Booleans	Truth-Values	Yes	Yes	—	$T *or* $F
Handles	Structures	Yes	Yes	—	'*STRUC*
Closures	Functions	Yes	No	CCONS	{closure}
Rails	Sequences	Some	No	RCONS	[*STRUC* ... *STRUC*]
Atoms	(Φ of Binding)	No	—	ACONS	*alphamerics*
Pairs	(Value of App.)	No	—	PCONS	(*STRUC* . *STRUC*)

Table 1: The 2-LISP (and 3-LISP) Categories

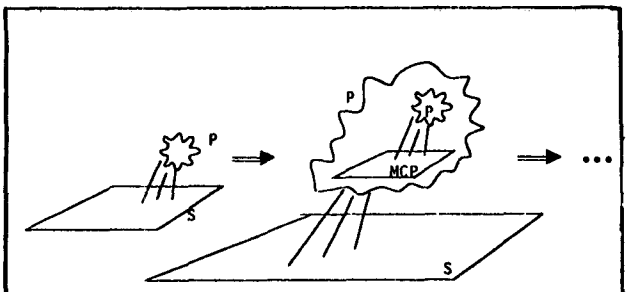


Figure 12: Meta-Circular Processors

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

More generally, 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,[6] it is possible to construct what are commonly known as *metacircular interpreters*, which we 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. They are circular for two reasons. First, they have to be run by that processor in order to yield any sort of behaviour (since they are *programs*, not *processors*, strictly). Second, the behaviour they would thereby engender can be known only if one knows beforehand what the processor does. (Standard techniques of fixed points, furthermore, are of no help in discharging this circularity, because this kind of modelling is a kind of self-mention, whereas recursive definitions are more self-use.) 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 12, showing how, if we ever replace P in Figure 1 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 (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) $\simeq$ P, where by '$\simeq$' we mean behaviourally equivalent in some appropriate sense. The equivalence is, of course, a global equivalence; 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. The point, rather, 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 directly by P. MCPs 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" we 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. (I will 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.)

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 don't traffic in such entities.) 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 in [McCarthy 1965] and Steele and Sussman's versions for Scheme (see for example [Sussman and Steele 1978b]), it is absorbed. Two versions of the 2-Lisp metacircular processor, one absorbing and one making explicit the continuation structure, are presented in Figures 13 and 14. Note, however, that in both cases the underlying agency or *anima* is not reified; it remains entirely absorbed by the processor of the MCP. We have no mechanism to designate a process (as opposed to structures), and no method of obtaining causal access to an independent locus of active agency (the reason, of course, being that we have no theory of what a process is).

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 were 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 was not run by the primitive processor, but was run by the primitive processor running another copy of the MCP. Etc. 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 we call a *reflective tower*, defined as 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, as $n \to \infty$, of 2-Lisp/n.

We will look at the revised MCP presently, but some general properties of this tower architecture can be pointed out first. A rough idea of the levels of processing is given in Figure 15: 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. The implied infinite regress is not problematic, and 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, one can think of there being a primitive and noticeable *reflective act*, which causes the processor to shift levels rather markedly (this is the explanation that best coheres with some of our pre-theoretic intuitions about reflective thinking in the sense of contemplation). On the other hand, the explanation

```
(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! env)
                      (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!)))))
```

Figure 13: A Non-Continuation-Passing 2-LISP MCP

given in the previous paragraph leads one to think of an infinite number of levels of reflective processors, each implementing the one below.[7] On such a view it is not coherent either to ask at which level the tower is running, or to ask how many reflective levels are running: in some sense they are all running at once. Exactly 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, being interior to the other, supplies the anima or agency of the outer one. 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 don't have concurrency; it is more a part/whole kind of relation. 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. We will not take a principled view on which account — a single locus of agency stepping between levels, or an infinite hierarchy of simultaneous processors — is correct, since they turn out to be behaviourally equivalent. (The simultaneous infinite tower of levels is often the better way to understand processes, whereas a shifting-level viewpoint is sometimes the better way to understand programs.)

3-Lisp, as we said, is an infinite reflective tower based on 2-Lisp. The code at each level is like the continuation-passing 2-Lisp MCP of Figure 14, but extended 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 located in a reflective tower, we call this code the 3-Lisp *reflective processor*). One gains this access by using what are called *reflective procedures* — procedures that, when invoked, are run not at the level at which the invocation occurred, but one level higher, at the level of the reflective processor running the program, given as arguments those structures being passed around in the reflective 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 [strc env cont]
    (cond [(normal strc) (cont strc)]
          [(atom strc) (cont (binding strc env))]
          [(rail strc) (normalise-rail strc env cont)]
          [(pair strc) (reduce (car strc) (cdr strc) env cont)])))
(define REDUCE
  (lambda simple [proc args env cont]
    (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! env cont)
                    (expand-closure proc! args! cont))))]
          [intensional
            (if (primitive proc!)
                (reduce-primitive-intensional
                  proc! *args env cont)
                (expand-closure proc! *args cont))]
          [macro (expand-closure proc! *args
                   (lambda simple [result]
                     (normalise result env cont)))])))))
(define NORMALISE-RAIL
  (lambda simple [rail env cont]
    (if (empty rail)
        (cont (rcons))
        (normalise (1st rail) env
          (lambda simple [first!]
            (normalise-rail (rest rail) env
              (lambda simple [rest!]
                (cont (prep first! rest!)))))))))
(define EXPAND-CLOSURE
  (lambda simple [proc! args! cont]
    (normalise (body proc!)
               (bind (pattern proc!) args! (env proc!))
               cont)))
```

Figure 14: A Continuation-Passing 2-LISP MCP

Reflective procedures are essentially analogues of subroutines to be run "in the implementation", except that they are in the same dialect as that being implemented, and can use all the power of the implemented language in carrying out their function (e.g., reflective procedures can themselves use reflective procedures, without limit). There is not a tower of different languages — there is a single dialect (3-Lisp) all the way up.

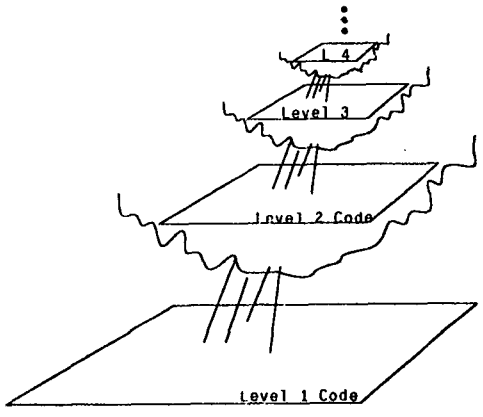


Figure 15: The 3-LISP Reflective Tower

Rather, there is a tower of processors, necessary because there is different processor state at each reflective level.

Some simple examples will illustrate. Reflective procedures are "defined" (in the sense we 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" overview in Figure 16, including a working 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, 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; LAMBDA, IF, and QUOTE are all non-primitive (user) definitions in 3-Lisp, again illustrated in the insert. There is also a simplistic break package, 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 importance of these examples comes from the fact that they are causally connected in the right way, and will therefore 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. There is also a 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 can be understood only with a close inspection of the 3-Lisp reflective processor (Figure 17), the promised modification of the continuation-passing 2-Lisp metacircular processor mentioned above. NORMALISE (line 7) takes an structure, environment, and continuation, returning the structure unchanged (i.e., sending it to the continuation) if it is in normal form, looking up the binding if it is an atom, normalising the elements if it is a rail (NORMALISE-RAIL is 3-Lisp's tail-recursive continuation-passing analogue of Lisp 1.5's EVLIS), and otherwise reducing the CAR (procedure) with the CDR (arguments). REDUCE (line 13) first normalises the procedure, with a continuation (C-PROC!) that checks to see whether it is reflective (by convention, we use exclamation point suffixes on atom names used as variables to designate normal form structures). If it is not reflective, C-PROC! normalises the arguments, with a continuation that either expands the closure (lines 23–25) if the

Figure 16: 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 it is semantically identical to the simple numeral:

```
(define THREE
   (lambda reflect [[] env cont]
      (cont '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 cont]
      (cont ↑(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":

```
(define QUIT
   (lambda reflect [[] env cont]
      '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-cont]
    (normalise body
       (bind tag
             ↑(lambda reflect [[answer] throw-env throw-cont]
                 (normalise answer throw-env catch-cont))
             catch-env)
       catch-cont)))
```

For example:

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

would designate seventeen and return the numeral 17.

In addition, 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, with the following definitions:

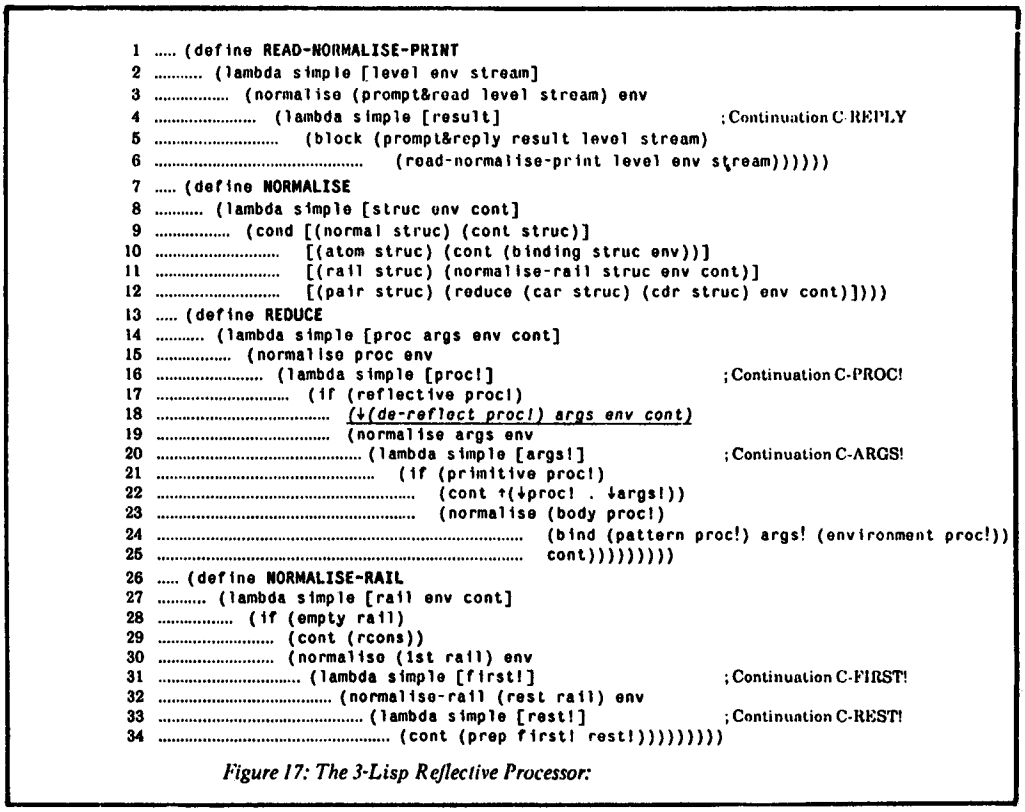
```
(define LAMBDA
  (lambda reflect [[kind pattern body] env cont]
    (cont (ccons kind ↑env pattern body))))

(define IF
  (lambda reflect [[promise then else] env cont]
    (normalise premise env
      (lambda simple [premise!]
        (normalise (ef ↓premise! then else) env cont)))))

(define QUOTE
  (lambda reflect [[arg] env cont] (cont ↑arg)))
```

Some comments. First, the definition of LAMBDA just given is of course circular; a non-circular but effective version is given in Smith and des Rivières [1984]; the one given in the text, 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 only one of the second two, depending on the result of normalising the first. Finally, 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 [[arg] env cont]
    (block (print arg primary-stream)
           (read-normalise-print ">>"
              (bind* ['env ↑env]
                     ['cont ↑cont]
                     ['return ↑(lambda reflect [[a2] 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.



```
1  ..... (define READ-NORMALISE-PRINT
2  ..... (lambda simple [level env stream]
3  ..... (normalise (prompt&read level stream) env
4  ..... (lambda simple [result]                          ;Continuation 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!]                           ;Continuation C-PROC!
17 ..... (if (reflective proc!)
18 ..... (↓(de-reflect proc!) args env cont)
19 ..... (normalise args env
20 ..... (lambda simple [args!]                           ;Continuation 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!]                           ;Continuation C-FIRST!
32 ..... (normalise-rail (rest rail) env
33 ..... (lambda simple [rest!]                           ;Continuation C-REST!
34 ..... (cont (prep first! rest!)))))))))
```

Figure 17: The 3-Lisp Reflective Processor.

procedure is non-primitive, or else directly executing it if it is primitive (line 22).

Consider (REDUCE '+ '[X 3] ENV ID), for example, where X is bound to the numeral 2 and + to the primitive addition closure in ENV. At the point of line 22, PROC! will designate that primitive closure, and ARGS! will designate the normal-form rail [2 3]. Since addition is primitive, we must simply *do* the addition. (PROC! . ARGS!) won't work, because PROC! and ARGS! are at the wrong level; they designate structures, not functions or arguments. So, for a brief moment, we de-reference them (with ↓), do the addition, and then regain our meta-structural viewpoint with the ↑.[8] If the procedure is reflective, however, it is (as shown in line 18 of Figure 17) called directly, not processed, and given the obvious three arguments (ARGS, ENV, and CONT) that are being passed around. The ↓(DE-REFLECT PROC!) is merely a mechanism to purify the reflective procedure so that it doesn't reflect again, and to de-reference it to be at the right level (we want to use, not mention, the procedure that is 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.

Line 18 is the essence of 3-Lisp; it 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. The 3-Lisp claim is that an exactly equivalent reflective processor can be processing this code, without vicious threat of infinite ascent.

A reflective procedure, in sum, arrives in the middle of the processor context. It is handed environment and continuation structure 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 in turn is represented in structures passed around by the processor one level above it. Thus it is given causal access to the state of the process that was in progress (answering one of our initial requirements), and it can of course 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 conflict with the code being run below it.

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 3-Lisp tower, we are suggesting, provides just the right balance between these two extremes, solving the problem of vantage point as well as of causal connection.

The 3-Lisp reflective processor unifies three traditionally independent capabilities in Lisp: the explicit availability of EVAL and APPLY, the ability to support metacircular processors, and explicit operations (like Maclisp's RETFUN and Interlisp's FRETURN) for debugging purposes. 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.

9. The Threat of Infinity, and a Finite Implementation

The argument as to why 3-Lisp is finite is complex in detail, but simple in outline and in substance. Basically, one shows that the reflective processor is fully tail-recursive, in two senses: a) 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 b) it itself is fully tail-recursive, in the sense that all recursive calls within it (except for unimportant subroutines) occur in tail-recursive position. The reflective processor, can be executed by a simple finite state machine. In particular, it can run itself without using any state at all. Once the limiting behaviour of an infinite tower of copies of this processor is determined, therefore, that entire chain of processors can be simulated by another state machine, of complexity only moderately greater than that of the reflective processor itself. (It is an interesting open research question

whether that "implementing" processor can be algorithmically derived from the reflective processor code.) A full copy of such an implementing processor — about 50 lines of 2-Lisp — is provided in [Smith and des Rivières 1984]; a more substantive discussion of tractability will appear in [Smith forthcoming].

10. Conclusions and Morals

Fundamentally, the use of Lisp as a language in which to explore semantics and reflection is of no great consequence; the ideas should hold in any similar circumstance. We chose 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 for a mathematical study of reflection).

Furthermore, the technique we used in defining 3-Lisp can be generalised rather directly to these other languages. In order to construct a reflective dialect one needs a) to formulate a theory of the language analogous to the metacircular processor descriptions we have examined, b) to embed this theory within the language, and c) to connect the theory with the underlying language in a causally connected way, as we did in line 18 of the reflective processor, by providing reflective procedures invocable in the object language but run in the processor. It remains, of course, to implement the resulting infinite tower; a discussion of general techniques is presented in [desRivières, forthcoming].

It is partly a consequence of using Lisp that we 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 modifying operations would be supported (changing bindings is one, but one might wish to excise bindings completely, splice new ones in, etc.). In standard λ-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, we assume in 3-Lisp that the main structural field is simply visible from all code; one could define an alternative dialect in which the field, too, was threaded through the processor as an explicit argument, as in standard metatheory.

The representation of procedures as closures is troublesome (indeed, closures are failures, in the sense that they encode far more information than would be required to identify a function in intension; the problem being that we don't yet know what a function in intension might be.). 3-Lisp unarguably provides far too fine-grained (i.e., metastructural) access to function designators, including continuations, and the like. Given an 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. We did not follow this direction here only to avoid taking on another very difficult problem, but we will move in this direction in future work.

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. It is all a question of what you want to make explicit, and what you want to absorb. 3-Lisp, as currently defined, reifies the environment and continuation, making explicit what was implicit one level below. It absorbs the structural field (and partly absorbs the global environment); as mentioned earlier, it completely absorbs the animating agency of the whole computation. If one defines a reflective processor based on a metacircular processor that also absorbs the representation of control (i.e., like the MCP in Figure 13, which uses the control structure of the processor to encode the control structure of the code being processed), then reflective procedures could not affect the control structure. In any real application, it would need to be determined just what parts of the underlying dialect required reification. One could perhaps provide a dialect in which a reflective procedure could specify, with respect to a very general theory, what aspects it wanted to get explicit access to. Then operations, for example, that needed only environment access, like BOUND, could avoid having to traffic in continuations.

A final point. I have talked throughout about semantics, but have presented no mathematical semantical accounts of any of these dialects. To do so for 2-Lisp is relatively straightforward (see Smith [forthcoming]), but I have not yet worked out the appropriate semantical equations to describe 3-Lisp. It would be simple to model such equations on the implementation mentioned in section 9, but to do so would be a failure: rather, one should instead take the definition of 3-Lisp in terms of the infinite virtual tower (i.e., take the limit of 2-Lisp/n), and then prove that the implementation strategies of section 9 are correct. This awaits further work. In addition, I want to explore what it would be to deal explicitly, in the semantical account, with the anima or agency, and with the questions of causal connection, that are so crucial to the success of any reflective architecture. These various tasks will require an even more radical reformulation of semantics than has 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 Xerox PARC, as part of the Situated Language Program of Stanford's Center for the Study of Language and Information.

Notes

1. See [Doyle 1980], [Weyrauch 1980], [Genesereth and Lenat 1980], and [Batali 1983].
2. In the dialects we consider, the metastructural capability must be provided by primitive quotation mechanisms, as opposed to merely by being able to model or designate syntax — something virtually any calculus can do, using Gödel numbering, for example — for reasons of causal connection.
3. Most programming languages, such as Fortran and Algol 60, are neither higher-order nor metastructural; the λ-calculus is the first but not the second, whereas Lisp 1.5 is the second but not the first (dynamic scoping is a contextual protocol that, coupled with the meta-structural facilities, partially allows Lisp 1.5 to compensate for the fact that it is only first-order). At least some incarnations of Scheme, on the other hand, are both (although Scheme's metastructural powers are limited). As we will see, 2-Lisp and 3-Lisp are very definitely both metastructural and higher-order.
4. For what we 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 we will turn in a moment). It is for such languages, in particular, that the composition $\Phi \circ \mathrm{O}$, which we might call Φ', would be formulated. And this, for obvious reasons, is what 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. Thus, for example, Φ' in logic would be the interpretation function of standard model theory. In what we will call *computational* languages, on the other hand, questions of processing do arise.
5. 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.
6. Virtually any language, of course, has the requisite power to do this kind of modelling. In a language with meta-structural abilities, the meta-circular processor can represent programs for the MCP *as themselves* — this is always done in Lisp MCPs -- but we need not define that to be an essential property. The term 'metacircular processor' is by no means strictly defined, and 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, because it *does* have the requisite

causal connections, and therefore an essential part of the 3-Lisp architecture.

7. Curiously, there are also intuitions about contemplative thinking, where one is both detached and yet directly present, that fit more with this view.
8. 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 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 the processor running that processor will also come racing towards the edge of the same cliff, and will similarly duck responsibility, handing the primitive up yet another level. In fact every primitive ever executed is handed all the way to the top of the tower. There is 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 the stonehenge and pyramids, reaching down through a long tunnel at just one particular moment during the year). 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 to the level below. All infinitely fast.

References

Batali, J., "Computational Introspection", M.I.T. Artificial Intelligence Laboratory Memo AIM-TR-701 (1983).

desRivières, J. "The Implementation of Procedurally Reflective Languages", (forthcoming).

Doyle, J., *A Model for Deliberation, Action, and Introspection*, M.I.T. Artificial Intelligence Laboratory Memo AIM-TR-581 (1980).

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

Genesereth, M., and Lenat, D. B., "Self-Description and -Modification in a Knowledge Representation Language", Heuristic Programming Project Report HPP-80-10, Stanford University CS Dept., (1980).

McCarthy, J. et al., *LISP 1.5 Programmer's Manual*. Cambridge, Mass.: The MIT PRess (1965).

Smith, B., *Reflection and Semantics in a Procedural Language*, M.I.T. Laboratory for Computer Science Report MIT-TR-272 (1982).

Smith, B. and desRivières, J. "Interim 3-LISP Reference Manual", Xerox PARC Report CIS-*nn*, Palo Alto (1984, forthcoming).

Steele, G., "LAMBDA: The Ultimate Declarative", M.I.T. Artificial Intelligence Laboratory Memo AIM-379 (1976).

Steele, G., and Sussman, G. "The Revised Report on SCHEME, a Dialect of LISP", M.I.T. Artificial Intelligence Laboratory Memo AIM-452, (1978a).

Steele, G., and Sussman, G. "The Art of the Interpreter, or, The Modularity Complex (Parts Zero, One, and Two)", M.I.T. Artificial Intelligence Laboratory Memo AIM-453, (1978b).

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