Interim 3-LISP Reference Manual

Brian Cantwell Smith and Jim des Rivières

June 1984

Intelligent Systems Laboratory
ISL-1

Corporate Accession P8400004

The paper that appears in Appendix C is printed with the kind permission of the Association for Computing Machinery.

XEROX

PALO ALTO RESEARCH CENTER
3333 Coyote Hill Road
Palo Alto, California 94304

INTERIM 3-LISP REFERENCE MANUAL

Brian C. Smith
Jim des Rivières

Intelligent Systems Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, California, 94304; and

Center for the Study of Language and Information
Stanford University, Stanford, CA 94305

The 3-LISP Reflective Processor

```
(define READ-NORMALIZE-PRINT
  (lambda simple [level env stream]
    (normalize (prompt&read level stream) env
      (lambda simple [result]                        ; Continuation C-REPLY
        (block (prompt&reply result level stream)
               (read-normalize-print level env stream))))))

(define NORMALIZE
  (lambda simple [exp env cont]
    (cond [(normal exp) (cont exp)]
          [(atom exp) (cont (binding exp env))]
          [(rail exp) (normalize-rail exp env cont)]
          [(pair exp) (reduce (car exp) (cdr exp) env cont)])))

(define REDUCE
  (lambda simple [proc args env cont]
    (normalize proc env
      (lambda simple [proc!]                         ; Continuation C-PROC!
        (if (reflective proc!)
            (↓(de-reflect proc!) args env cont)
            (normalize args env
              (lambda simple [args!]                 ; Continuation C-ARGS!
                (if (primitive proc!)
                    (cont ↑(↓proc! . ↓args!))
                    (normalize (body proc!)
                               (bind (pattern proc!) args! (environment proc!))
                               cont))))))))

(define NORMALIZE-RAIL
  (lambda simple [rail env cont]
    (if (empty rail)
        (cont (rcons))
        (normalize (1st rail) env
          (lambda simple [first!]                    ; Continuation C-FIRST!
            (normalize-rail (rest rail) env
              (lambda simple [rest!]                 ; Continuation C-REST!
                (cont (prep first! rest!)))))))))
```

Table of Contents

Acknowledgments

The authors would like to thank the many individuals that have contributed in various ways to the development of 3-LISP and the ideas that underlie it, especially Austin Henderson, Hector Levesque, Mike Dixon, Greg Nuyens, Dan Friedman, Gumby, Henry Thompson, and any other members of The Knights of the Lambda Calculus, wherever they may be.

We are pleased to follow the honorable tradition of developing LISP dialects and meta-circular LISP interpreters — a tradition that owes much to John McCarthy, Gerry Sussman, Guy Steele, and John Allen.

The research was conducted in the Intelligent Systems Laboratory at Xerox PARC, as part of the Situated Language Program of Stanford's Center for the Study of Language and Information. We would particularly like to thank John Seely Brown for his initial and continuing support of this project.

The paper that appears in Appendix C is printed with the kind permission of the Association for Computing Machinery.

1. Introduction

The 3-LISP programming language is designed to illustrate both the integration of declarative and procedural semantics in a unified calculus, and the provision of reflective capabilities. It is a direct descendant of McCarthy's LISP 1.5 [McCarthy 65] and of Sussman and Steele's SCHEME [Sussman&/Steele 75, 76a, 76b, 77, 78a, 78b]; many features of those formalisms are embodied in 3-LISP without comment. There are, however, some major differences between 3-LISP and prior LISP dialects. More specifically:

1. ***Statically Scoped and Higher Order:*** Like SCHEME and the untyped λ-calculus, but unlike LISP 1.5, functions of any degree may be used in 3-LISP as arbitrary arguments. Free variables are statically (lexically) scoped. Consequently, flat (non meta-structural) 3-LISP is on its own a higher-order functional calculus.
2. ***Untyped and Unsorted:*** Though the semantic domain and primitive functions are typed (in the computer scientist's sense — what logicians call *sorted*), user-defined procedures need not be typed, and no typing information is explicitly stated. In addition, there are no type restrictions (in the logician's sense).
3. ***Meta-Structural:*** As in all LISPs, quotation is provided primitively, enabling the explicit mention of program structures. Because naming and normalization are orthogonal, quotation is a *structural* primitive, not a *functional* primitive (i.e., there is no primitive QUOTE procedure). *Handles* (normal-form designators of internal structures) are unique and canonical normal-form structure designators.
4. ***Semantically Rationalized:*** Traditional evaluation is rejected in favor of independent notions of *simplification* and *designation*. The 3-LISP processor is based on a form of simplification called *normalization* that takes each structure into a co-designating structure in normal form. As a consequence, processing is *semantically flat:* programs may cross the meta-structural hierarchy only with the explicit use of the two level-crossing primitives (↑ and ↓, *q.v.*); note that reflective procedures are *not* level-crossing. In addition, the processor is *idempotent* (all normal-form structures normalize to themselves). With the exception of the one side-effect primitive (REPLACE), the declarative (Φ) and procedural (Ψ) semantics can be specified independently.
5. ***Category Aligned:*** There is a one-to-one correspondence across primitive structural categories, declarative semantic categories, and categories of procedural treatment. In addition, there are corresponding notational categories, although the standard notation (see §3) has some slight additional complexity for user convenience.
6. ***Procedures and Functions:*** The standard (but user-defined) procedure LAMBDA is used purely as a naming operator; recursion is treated with the explicit use of circular-structure generating Y-operators. *Closures* are a distinguished structural category, and are normal-form function designators.
7. ***Procedurally Reflective:*** 3-LISP supports two kinds of procedures: *simple* and *reflective*. Reflective procedures are run not in the object level of a program, but integrated into an explicit version of the processor that was running that program. Thus, the 3-LISP virtual machine consists of an infinite tower of type-equivalent processors. This architecture unifies the traditional notions of an explicitly available EVAL and APPLY, meta-circular interpreters, and idiosyncratic extensions to facilitate debugging.

1.a. The 3-LISP Experiment

Smith's dissertation [Smith 82a] contains a detailed justification of the design principles underlying 3-LISP. The language described in this manual is fundamentally the same as the original version, though it has undergone some evolution (e.g., closures now have their own structural category). Similarly, the techniques used to implement 3-LISP are basically those discussed in chapter 5 of the dissertation, but with numerous refinements in order to achieve reasonable performance. A summary paper that appeared in the 1984 ACM Principles of Programming Languages Conference Proceedings [Smith 84a] is included as an appendix.

3-LISP is an experimental language — an experiment still in its early stages. There are active investigations on three fronts: the formal semantics of reflection; the development of a reflective language supporting data abstraction; and 3A-LISP, a dialect of 3-LISP free of side effects.

Be that as it may, it was felt that sufficient progress had been made to warrant making available this interim reference manual, which describes an implementation, again interim, built on top of INTERLISP-D and running on Xerox 1100 series machines. The authors welcome any and all comments on the manual, on the language, or, more generally, on the concepts of reflection and semantic rationalization.

1.b. Organization of this Manual

The goal of this manual is to provide someone with enough information to be able to understand and use the INTERLISP-D based implementation of 3-LISP. §2 is with a primer on the 3-LISP language and reflective programming. This is followed in §3 by a detailed summary of the structural field and standard notation. The standard procedures of 3-LISP are documented in §4 (the 3-LISP code for all non-primitive standard procedures can be found in Appendix A). §5 contains instructions on how to use the INTERLISP-D based implementation of the system.

Of special interest to implementers, a sketch of how one might go about implementing 3-LISP is presented in Appendix B.

This manual assumes familiarity with [Smith 84a], which explains the philosophy underlying the design of 3-LISP and introduces the concepts and terminology used to explain the system; this paper is reprinted in Appendix C. While in one sense it is true that 3-LISP is merely a distillation of existing computational practice as adhered to by the LISP community, it is also true that 3-LISP departs rather radically from some of the fundamental notions and terms (such as *evaluation*) upon which LISP is based. For this reason, Appendix C will be worthwhile preparation for even the experienced LISP hacker.

2. A 3-LISP Primer

3-LISP can claim to be a dialect of LISP only on a generous interpretation. It is unarguably *more different from the original LISP 1.5 than is any other dialect that has been proposed*, including, for example, SCHEME [Sussman&/Steele 75, 76a, 76b, 77, 78a, 78b], MDL [Galley&Pfister 75], NIL [White 79], MACLISP [Moon 74], INTERLISP [Teitelman 78], COMMON LISP [Steele et al. 82], and T [Rees 82].

In spite of this difference, however, it is important to our enterprise to call this language LISP. We do not simply propose it as a new variant in a grand tradition, perhaps better suited to a certain class of problems than those that have gone before. Rather, we claim that the architecture of this new dialect, in spite of its difference from that of standard LISPs, *is a more accurate reconstruction of the underlying coherence that organizes our communal understanding of what LISP is*. We are making a claim, in other words — a claim that should ultimately be judged as right or wrong. Whether 3-LISP is *better* than previous LISPs is, of course, a matter of some interest on its own, but it is not the principle motivation behind its development.

This section is tutorial in nature; §2.a. introduces the basic 3-LISP language, leaving details of the reflective processor and reflective procedures to §2.b. Details of the structural field, standard notation, and the standard procedures are covered in subsequent sections.

2.a. The Basic Language

Perhaps the best way to begin to understand a new programming language is to watch it in action. Better still is seeing it put through its paces and getting a running commentary to boot. So, without further ado, let's dive right in and play.

```
1> 100
1= 100
```

The ground rules for these interactions with the 3-LISP system are straight-forward. The system usually prompts with '1>'. Shown in italics following the system prompt is our input just as we typed it — in this case '100'. The system's reply to our input is shown on the following line, right after the '1=' marker. In this case, the answer was '100'. The correct way to view the system is that it accepts an expression, simplifies it, and then displays the result. Since the expression 100 cannot be further simplified, the system just spits it back at us. Both the original input and the result designate the abstract number one hundred.

```
1> (+ 2 3)
1= 5
```

The expression '(+ 2 3)' is the 3-LISP way of saying "the value of applying the addition function to the numbers two and three." The system answers five because that is exactly what this fancy name-for-a-number amounts to. Again, we are seeing that a) both the input and the output expression designate the same object, and b) the answer is in its simplest possible form. Expressions enclosed in '(' and ')' are called pairs (occasionally, redexes) and are taken to designate the value of applying the function designated by the first sub-expression to the arguments designated by the remaining sub-expressions. Names like '+' are called atoms; what they designate depends on where they are used. In all of these examples, their meaning is the standard one supplied by the off-the-shelf 3-LISP system; not surprisingly, '+' designates the function that adds numbers together.

```
1> (+ 2 (* 3 (+ 4 5)))
1= 29
```

There is no limit on how complicated the input expressions can be. The last one can be read "two plus three times the sum of four and five," namely twenty nine.

```
1> [1 2 3]
1= [1 2 3]
```

Structures notated by expressions enclosed in '[' and ']' are called rails, and designate the abstract sequence composed of the objects designated by the various sub-expressions in the order given. Thus [1 2 3] designates the abstract sequence of containing, in order, the numbers one, two, and three.

```
1> []
1= []
```

The empty sequence that contains no elements is designated [].

```
1> [(* 3 3) (* 4 4) (* 5 5)]
1= [4 9 16]
```

All complex sub-expressions are simplified in the process of deriving the answer.

```
1> [1 [2 (+ 1 2)] 4]
1= [1 [2 3] 4]
```

Moreover, rails may appear as sub-expressions inside other rails, making it possible to refer to sequences comprised of numbers and other sequences.

```
1> (1ST [1 2 3])
1= 1
1> (REST [1 2 3])
1= [2 3]
1> (PREP (+ 99 1) [1 2 3])
1= [100 1 2 3]
1> (LENGTH [1 2 3])
1= 3
```

The standard operations on sequences are: 1ST — for the first component of a (non-empty) sequence; REST — for the sequence consisting of every element but the first; PREP — for the sequence consisting of the first argument *prep*ended to the second argument; LENGTH — for the number of elements in the sequence; and plenty more (all explained in §4).

```
1> (= 2 2)
1= $T
1> (= 2 (+ 1 2))
1= $F
1> (= $T $F)
1= $F
```

The booleans $T and $F are the standard designators of Truth and Falsity, respectively.

```
1> (IF $T (+ 2 2) (- 2 2))
1= 4
1> (IF $F (+ 2 2) (- 2 2))
1= 0
1> (IF (< -100 0) -1 1)
1= -1
1> (IF (ZERO 0) (= 1 2) 13)
1= $F
```

Redexes that designate truth values play an important role in IF expressions, which are used to choose between their last two arguments based on the truth of the first argument. Also important, and unlike the other standard procedures we have discussed so far, IF does not process all of its arguments — the argument that is not selected is ignored completely. In contrast, most standard procedures always begin by processing *all* of their arguments (i.e., for the most part, 3-LISP is an applicative-order language); we call such procedures *simple*. Thus IF is simply not simple. (Although we will see later that IF is not really a magic keyword, no real harm will come from thinking of it that way).

```
1> +
1= {simple + closure}
1> 1ST
1= {simple 1ST closure}
1> IF
1= {reflective IF closure}
```

To summarize what we have seen so far: numerals, like '10', are used to designate numbers; the two booleans $T and $F are used to designate truth values; atoms, like 'PREP', are used as variables that take their meaning from the context in which they are used (so far, this has been the standard global context); rails are used to designate abstract sequences; and pairs designate the value of applying a function to some arguments. Also, there are as-yet-unexplained structures called closures that appear to serve as function designators. As it turns out, these are the basic building blocks on which the 3-LISP tower is erected.

The standard 3-LISP system comes with over 140 standard procedures (see §4) and an abstraction facility that allows existing procedures to be combined to form new ones.

```
1> (LAMBDA SIMPLE [X] (* X X))
1= {closure}
1> ((LAMBDA SIMPLE [X] (* X X)) 10)
1= 100
1> ((LAMBDA SIMPLE [X] (* X X)) (+ 3 3))
1= 36
1> ((LAMBDA SIMPLE [A B C] (= (* C C) (+ (* B B) (* A A)))) 3 4 5)
1= $T
```

LAMBDA expressions have three parts: a procedure type (normally SIMPLE; later we shall see others); a list of parameter names (more generally, a parameter pattern); and a body. In the usual case of SIMPLE lambda expression, the new function designated by the LAMBDA redex can be computed by processing the body of the expression in the context in which the parameters are bound to the (already simplified) arguments. Variables not mentioned in the parameter pattern take their values from the context surrounding the LAMBDA redex (i.e., 3-LISP's functional abstraction mechanism is statically, lexically, scoped like PASCAL, SCHEME, and the λ-calculus, but unlike APL and standard LISPs).

```
1> ((LAMBDA SIMPLE [F] (F 10 10)) +)
1= 20
1> ((LAMBDA SIMPLE [F] (F 10 10)) *)
1= 100
1> ((LAMBDA SIMPLE [F] (F 10 10)) =)
1= $T
1> (DEFINE CONSTANT
      (LAMBDA SIMPLE [M]
         (LAMBDA SIMPLE [JUNK] M)))
1= 'CONSTANT
1> (CONSTANT 10)
1= {closure}
1> ((CONSTANT 10) 1)
1= 10
1> ((CONSTANT 10) 100)
1= 10
1> ((LAMBDA SIMPLE [F] (F F)) (LAMBDA SIMPLE [F] (F F)))
```

[N.B.: We're still waiting for the system's ruling on this one!]

Moreover, functions are first-class citizens, along with numbers, truth values, and sequences. They can be passed as arguments to, and returned as the result of, other functions (i.e., 3-LISP is a higher-order functional calculus).

```
1> ((LAMBDA SIMPLE [A B C] (+ A (* B C))) 1 2 3)
1= 7
1> ((LAMBDA SIMPLE [A B C] (+ A (* B C))) . [1 2 3])
1= 7
```

Although it is usually not convenient to be so picky, it is true that every procedure takes but a single argument, which is, in turn, usually a sequence. The notation for pairs that we have been writing all along is just short for the "dot" notation illustrated above.

```
1> ((LAMBDA SIMPLE X X) . 10)
1= 10
1> ((LAMBDA SIMPLE X X) 1 2 3)
1= [1 2 3]
1> (SET W [4 5 6])
1= 'OK
1> ((LAMBDA SIMPLE X X) . W)
1= [4 5 6]
1> ((LAMBDA SIMPLE X X) W)
1= [[4 5 6]]
1> ((LAMBDA SIMPLE [X Y Z] [X Y Z]) . W)
1= [4 5 6]
```

When the parameter pattern is simply a variable (as opposed to a rail), the single true argument is bound to the parameter variable without de-structuring. On the other hand (the more typical case), variables in the parameter list are paired up with corresponding components of the argument sequence.

```
1> ((LAMBDA SIMPLE [[A B] [C D]] [(+ A C) (+ B D)]) [1 2] [3 4])
1= [4 6]
```

And, naturally, parameter patterns can get as fancy as necessary.

```
1> (DEFINE DOUBLE
      (LAMBDA SIMPLE [X] (+ X X)))
1= 'DOUBLE
1> (DOUBLE 2)
1= 4
1> (DOUBLE (DOUBLE 4))
1= 16
1> (SET X 10)
1= 'OK
1> X
1= 10
1> (SET X (+ X 10))
1= 'OK
1> (+ X 5)
1= 25
```

DEFINE is used to associate a name with a newly-composed function. More generally, SET is used to (re-)establish the value of a variable as an arbitrary object, not necessarily a function. Neither SET nor DEFINE is simple; both have a noticeable and lasting effect on the designation of the specified variable (they have what we call an *environment side-effect*).

```
1> (INPUT PRIMARY-STREAM) X
1= #X
1> (INPUT PRIMARY-STREAM) (
1= #(
1> (OUTPUT #? PRIMARY-STREAM)
?
1= 'OK
1> (IF (= (INPUT PRIMARY-STREAM) #?)
       (OUTPUT #Y PRIMARY-STREAM)
       (OUTPUT #N PRIMARY-STREAM)) ?
Y
1= 'OK
```

Ignoring the single quote mark for the time being, we see that there are standard procedures that have a different form of side-effect, called *external world side-effects.* INPUT causes a single character to be read from the specified input stream (PRIMARY-STREAM); OUTPUT causes a single character to be printed on the specified output strream. The objects written '*#x*' are called *charats* (for lack of a better name) and are taken as designating individual characters.

```
1> (BLOCK
     (OUTPUT #Y PRIMARY-STREAM)
     (OUTPUT #e PRIMARY-STREAM)
     (OUTPUT #s PRIMARY-STREAM))
Yes
1= 'OK
```

Another non-simple standard procedure, BLOCK, is used to process several expressions in sequence — a feature that is handy when side-effects of one kind or another are being employed (and utterly useless if they're not).

```
1> (DEFINE LOOP
      (LAMBDA SIMPLE [N]
         (IF (= N 0)
             'DONE
             (LOOP (1- N)))))
1= 'LOOP
1> (LOOP 10)
1= 'DONE
1> (LOOP 1000000)
1= 'DONE
```

The point of the above is that the space required to carry out (LOOP N) is independent of N. This important property of how 3-LISP (and SCHEME) is implemented allows for a flexible style of function decomposition reminiscent of the use of GOTO statements in many procedural languages.

```
1> (DEFINE ITERATIVE-FACTORIAL
      (LAMBDA SIMPLE [N]
         (LABELS [[
            LOOP (LAMBDA SIMPLE [I R]
                    (IF (= I 0)
                        R
                        (LOOP (1- I) (* I R))))]]
            (LOOP N 1))))
1= 'ITERATIVE-FACTORIAL
1> (ITERATIVE-FACTORIAL 1)
1= 1
1> (ITERATIVE-FACTORIAL 4)
1= 24
```

ITERATIVE-FACTORIAL is an excellent example of how to write LISP PROGs and GOs in a purely functional style and get exactly the same space and time performance.

```
1> (DEFINE FACTORIAL
      (LAMBDA SIMPLE [N]
         (IF (= N 0)
             1
             (* N (FACTORIAL (1- N))))))
1= 'FACTORIAL
1> (FACTORIAL 1)
1= 1
1> (FACTORIAL 4)
1= 24
```

The "recursive" definition of FACTORIAL — a required part of every language's reference manual — completes our cursory look at the basic 3-LISP language.

2.b. Introduction to the 3-LISP Reflective Processor

As discussed in §2.a. the reflective processor program is a program, written in 3-LISP, that shows how one goes about processing 3-LISP programs. The first gap to bridge on the road to writing such a program is to settle on an internal representation for 3-LISP programs. We need the ability not only to *use* 3-LISP expressions but also to *mention* them. To this end, we introduce a new type of structure, called handles, to designate other internal structures. For example, whereas the expression (+ 2 2), when written in a 3-LISP program, designates the number four, the expression

'(+ 2 2) designates that 3-LISP program fragment. Similarly, '+ designates the *atom* +, which in turn designates the addition function; '2 designates the *numeral* 2, which designates the abstract number two.

```
1> (+ 2 2)
1= 4
1> '(+ 2 2)
1= '(+ 2 2)
1> (TYPE (+ 2 2))
1= 'NUMBER
1> (TYPE '(+ 2 2))
1= 'PAIR
1> (TYPE +)
1= 'FUNCTION
1> (TYPE '+)
1= 'ATOM
```

Indeed, for each of the types of abstract objects that can be designated by a 3-LISP expression, there is a corresponding internal structural type that designates it (see §3.a. for further details).

```
1> (TYPE 1)                     1> (TYPE '1)
1= 'NUMBER                      1= 'NUMERAL
1> (TYPE $T)                    1> (TYPE '$T)
1= 'TRUTH-VALUE                 1= 'BOOLEAN
1> (TYPE [1 2 3])               1> (TYPE '[1 2 3])
1= 'SEQUENCE                    1= 'RAIL
```

Pairs can be dissected with the CAR and CDR primitives. The PCONS primitive is used to build pairs.

```
1> (CAR '(+ 2 2))
1= '+
1> (CDR '(+ 2 2))
1= '[2 2]
1> (PCONS '+ '[2 2])
1= '(+ 2 2)
```

RCONS is used to create rails (sequence designators). LENGTH, 1ST, REST, PREP, etc., work on arguments that designate rails as well as sequences. Sequences and rails are known collectively as ***vectors***.

```
1> '[1 (+ 2 2) 3]
1= '[1 (+ 2 2) 3]
1> (TYPE '[1 (+ 2 2) 3])
1= 'RAIL
1> (1ST '[1 (+ 2 2) 3])
1= '1
1> (REST '[1 (+ 2 2) 3])
1= '[(+ 2 2) 3]
1> (PREP '1 '[(+ 2 2) 3])
1= '[1 (+ 2 2) 3]
```

The internal structures used to designate other internal structures are called ***handles***. Handles too have handles. The term ***meta-structural hierarchy*** refers to the collection of structures that designate other structures. The standard procedures UP and DOWN, which are usually abbreviated with the prefix characters '↑' and '↓', are used to explore this meta-structural hierarchy.

```
1> (TYPE ''(+ 2 2))
1= 'HANDLE
1> (TYPE ''+)
1= 'HANDLE
1> (TYPE '''''''''''''''1)
1= 'HANDLE
1> (UP 1)
1= '1
1> (UP (+ 2 2))
1= '4
1> ↑1
1= '1
1> ↑(+ 2 2)
1= '4
1> (DOWN '1)
1= 1
1> (DOWN '[1 2 3])
1= [1 2 3]
1> ↓'1
1= 1
1> ↓'[1 2 3]
1= [1 2 3]
1> ↓'A
{Error: You can't get down from an atom.}
```

To insure against forgetting which way is "up", simply remember that going up *adds* additional ''s to the printed representation of a structure. Also note that in contrast to most LISPs ''s don't "fall off" expressions, so to speak; this property is called ***semantical flatness.***

```
1> ↑1
1= '1
1> ↑↑1
1= ''1
1> ↑↑↑1
1= '''1
```

2.b.i. Normalization

Having defined an internal representation for 3-LISP program fragments, let us now take a closer look at exactly what it means to "process" them. Recall that the basic operating cycle of 3-LISP involves reading an expression, simplifying it, and printing the result. The "meat" of the cycle is the middle step that takes an arbitrary expression onto a simpler expression. This simplification process is constrained in two ways: a) the "after" expression must be, in some sense, in lowest terms, and b) both the "before" and "after" expressions must designate the same object. In 3-LISP, "lowest terms" is defined as being in *normal-form.* Consequently, the simplification process which converts an expression into a normal-form co-designating expression is called ***normalization.***

We define a structure to be in ***normal form*** iff it satisfies three criteria: it is ***context-independent,*** meaning that its semantics (both declarative and procedural) are independent of context; it is ***side-effect free,*** meaning that processing it will engender no side-effects; and it is ***stable*** meaning that it is self-normalizing. Of the nine 3-LISP structure types, six-and-a-half are in normal-form: the handles, charats, numerals, booleans, closures, streamers, and *some* of the rails (those whose constituents are in normal form). The standard procedure NORMAL is charged with the task of testing for normal-formedness.

```
1> (NORMAL 'A)
1= $F
1> (NORMAL '1)
1= $T
1> (NORMAL '(1 . 2))
1= $F
1> (NORMAL '[1 2 3])
1= $T
1> (NORMAL '[X Y Z])
1= $F
```

2.b.ii. The Reflective Processor

Our problem, then, is to characterize the normalization procedure; call it NORMALIZE. Recall that whereas a regular process will typically deal with abstractions like numbers, sequences, and functions, the reflective processor will traffic in the internal structures that make up programs; i.e., pairs, atoms, numerals, rails, etc. In other words, the reflective processor will run one level of designation further away from the real world than the program that the reflective processor runs. We expect, therefore, that this procedure NORMALIZE will take at least one argument — an argument that designates the internal structure to be normalized, and will return the corresponding normal-form co-designator. Our expectations are illustrated by the following:

```
1> 1                          1> (NORMALIZE '1)
1= 1                          1= '1
1> $T                         1> (NORMALIZE '$T)
1= $T                         1= '$T
1> [1 2 3]                    1> (NORMALIZE '[1 2 3])
1= [1 2 3]                    1= '[1 2 3]
1> (+ 2 2)                    1> (NORMALIZE '(+ 2 2))
1= 4                          1= '4
```

One minor problem: the meaning of atoms, such as +, is dependent on context, and we have made no allowance for anything along these lines. We will posit a second argument to NORMALIZE, an ***environment***, that will encode just such a context. An environment is a sequence of two-tuples of atoms and bindings; thus the environment designated by the 3-LISP structure [['A '3] ['UGHFLG '$T] ['PROC ''FOO]] contains bindings for three structures (A, UGHFLG, and PROC, bound respectively to the numeral 3, the boolean $T, and the handle 'FOO). Note that all well-formed environments contain bindings for only atoms, and all bindings are normal-form structures. If an environment contains more than one binding for the same variable, the leftmost one has precedence. GLOBAL is the standard name for the global environment, which contains bindings for all of the standard procedures such as +, 1ST, and IF.

```
1> +                          1> (NORMALIZE '+ GLOBAL)
1= {simple + closure}         1= '{simple + closure}
1> (+ 2 2)                    1> (NORMALIZE '(+ 2 2) GLOBAL)
1= 4                          1= '4
```

More generally, we can now consider normalizations with respect to environments other than the global environment. For example:

```
1> (NORMALIZE '[A B] [['A '1] ['B '2]])
1= '[1 2]
1> (NORMALIZE '(ADD A B) [['A '1] ['B '2] ['ADD ↑+]])
1= '3
1> (NORMALIZE '100 [])
1= '100
```

But be quite clear on one thing. GLOBAL designates the *real* global environment. Consequently, any changes made to it in the course of normalizing an expression will be "for real."

```
1> (NORMALIZE '(SET SMILE 1234) GLOBAL)
1= ''OK
1> SMILE
1= 1234
1> (NORMALIZE '(SET + *) GLOBAL)
1= ''OK
1> (+ 2 5)
1= 10
```

We are making progress. We have identified and made explicit the context, an important aspect of any model of the processing of 3-LISP programs. While it would be feasible to base a dialect on a reflective processor that only made explicit the environment, the resulting language would be limited to "well-behaved" control operators, like IF and LAMBDA; non-local exit operators, like QUIT and THROW, that do not exhibit simple flow of control would be beyond the realm of such a dialect. To allow maximum flexibility with respect to control flow, it is essential that this control flow information be *explicitly* encoded by structures within the reflective processor. The solution adopted in 3-LISP is analogous to the approach taken in the denotational semantics literature [Stoy 77, Gordon 79]. In addition to an environment, the reflective processor's state includes a ***continuation***. NORMALIZE will take a third argument, called the continuation, that designates a function that should be applied to the result of the normalization. Programs which explicitly encode control flow information in a continuation are said to be written in ***continuation-passing style*** (***CPS***). The pros and cons of CPS can be seen in the following two procedures: SUMMER sums a sequence in a fairly obvious way; CPS-SUMMER is a CPS version of the same procedure.

```
1> (define SUMMER
      (lambda simple [s]
         (if (empty s)
             0
             (+ (1st s) (summer (rest s))))))
1= 'SUMMER
1> (SUMMER [1 2 3])
1= 6

1> (define CPS-SUMMER
      (lambda simple [s cont]
         (if (empty s)
             (cont 0)
             (cps-summer (rest s)
                (lambda simple [r]
                   (cont (+ (1st s) r)))))))
1= 'CPS-SUMMER
1> (CPS-SUMMER [1 2 3] ID)
1= 6
```

The most important difference to note is that the call to SUMMER inside SUMMER is buried within a + redex, whereas the inner call to CPS-SUMMER is not. Instead of using the capabilities of the underlying processor to remember what's to be done after the inner SUMMER computation is complete, CPS-SUMMER arranges for all information necessary to proceed the computation to be packaged as the continuation and explicitly passed along. The result is greater flexibility — at the price of degraded perspicuity. For example, if it were suddenly decided that our summing function was to return -1 if any of the elements were negative, we could revise CPS-SUMMER without much difficulty:

```
1> (define CPS-SUMMER2
     (lambda simple [s cont]
       (cond [(empty s) (cont 0)]
             [(negative (1st s)) -1]
             [$T (cps-summer2 (rest s)
                    (lambda simple [r]
                      (cont (+ (1st s) r))))])))
1= 'CPS-SUMMER2
1> (CPS-SUMMER2 [1 2 -3 4 5] ID)
1= -1
```

However, the job of updating SUMMER, while not hard, is not quite as straight-forward.

```
1> (define SUMMER2
     (lambda simple [s]
       (cond [(empty s) 0]
             [(negative (1st s)) -1]
             [$T (let [[r (summer2 (rest s))]]
                    (if (negative r) -1 (+ (1st s) r)))])))
1= 'SUMMER2
1> (SUMMER2 [1 2 -3 4 5])
1= -1
```

In summary, NORMALIZE will be written in CPS because we want the 3-LISP reflective processor to *encode* an explicit theory of control rather than simply *engendering* one. Again, using ID, which designates the identity function, as the continuation to extract the answer, we expect NORMALIZE to behave as follows:

```
1> (NORMALIZE '[A B] [['A '1] ['B '2]] ID)
1= '[1 2]
1> (NORMALIZE '(+ A B) (APPEND [['A '1] ['B '2]] GLOBAL) ID)
1= '3
1> (NORMALIZE '100 [] ID)
1= '100
1> (NORMALIZE '(OUTPUT #* PRIMARY-STREAM) GLOBAL ID)
*
1= ''OK
1> (NORMALIZE '(SET SMILE $T) GLOBAL ID)
1= ''OK
1> SMILE
1= $T
```

2.b.iii. NORMALIZE

We can now begin to present the actual definition of NORMALIZE, and explain why it does the right thing. Its definition is as follows:

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

Within NORMALIZE, EXP designates the expression (an internal structure) being normalized; ENV, the environment (a sequence); and CONT, the continuation (a function of one argument, also an internal structure). NORMALIZE is little more than a dispatch on the type of EXP (the only glitch being that normal-form rails are not a category of their own): normal-form structures (numerals, booleans, closures, charats, streamers, and some rails) are self-normalizing and are therefore passed to the continuation without further fuss; atoms (i.e., variables) are looked up (BINDING's job) in the current environment and the resulting binding returned; pairs are dissected and farmed out to REDUCE; and the remaining non-normal-form rails are handed off to NORMALIZE-RAIL.

(Aside: It is natural enough to ask whether there could be a different reflective processor for 3-LISP. The answer is both yes and no. If what is meant is a different reflective processor for the dialect of 3-LISP documented in this manual, the answer would have to be no. But "no" in the sense that "a six letter word spelled l-a-m-b-d-a" cannot mean any word other than "lambda." 3-LISP not only gives the general shape to the language — it also spells everything out. Moreover, these details are not just a part of this reference manual — interesting reading for the human reader, but completely hidden from the view of any program (e.g., in the way the micro-code for your machine is). The details of how 3-LISP programs are processed are, upon reflection, matters of public record, so to speak, and any program can find this out if it cares to probe in the right places. *There are very few secrets in a reflective language.* On the other hand, it is quite easy to imagine all sorts of 3-LISP-*like* languages, each with their own reflective processor that differs in minor ways (or even major ones) for *the* 3-LISP reflective processor described in this manual. For example, the 3-LISP described in Smith's thesis is definitely not the same 3-LISP as we are talking about here. In summary, *for any particular dialect of a reflective language there can be but a single reflective processor; change anything whatsoever and you'll have a slightly different dialect.*

2.b.iv. NORMALIZE-RAIL

We'll dispense with NORMALIZE-RAIL next. The utter simplicity of NORMALIZE-RAIL is somewhat obscured by the CPS protocols. The following non-CPS version should help to make clear what is going on:

```
(define NORMALIZE-RAIL                    ; Demonstration model — not for actual use.
  (lambda simple [rail env]
    (if (empty rail)
        (rcons)
        (prep (normalize (1st rail) env)
              (normalize-rail (rest rail) env)))))
```

Better still:

```
(define NORMALIZE-RAIL                ; Demonstration model — not for actual use.
   (lambda simple [rail env]
      (map (lambda simple [element] (normalize element env)) rail)))
```

NORMALIZE-RAIL simply constructs a rail whose components are the normal-form designators resulting from the normalizations of the compenents of the original rail. Although not explicit in the above, the components should be processed in left-to-right order. Lines 26-34 of the reflective processor spell out the details of the actual version of NORMALIZE-RAIL:

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

The two standard continuations (actually, continuation *families*), called C-FIRST! and C-REST!, correspond to intermediate steps in the normalization of a non-empty rail. C-FIRST! accepts the normalized first element in a rail fragment, and initiates the normalization of the rest of the rail. C-REST! accepts the normalized tail of a rail fragment, and is responsible for appending it to the front of the normalized first element.

2.b.v. REDUCE

We are now ready to tackle REDUCE, whose responsibility is to normalize pairs. As might be expected, REDUCE is the soul of the reflective processor — all sorts of interesting things go on with its confines.

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

There are basically three different ways of processing pairs: one way for non-primitive simple procedures (lines 23-25), one for the primitives (line 22), and one for what are called *reflective procedures* (line 18). We can isolate and study each of these cases one at a time, and free from the obscurity introduced by CPS. The first case is essentially:

```
(define REDUCE-NON-PRIMITIVE-SIMPLES          ; Demonstration model — not for actual use.
   (lambda simple [proc args env]
      (let [[proc! (normalise proc env)]
            [args! (normalise args env)]]
         (normalize (body proc!)
                    (bind (pattern proc!) args! (environment proc!))))))
```

Here we see that both the procedure, PROC, and the arguments, ARGS, are normalized in the current environment. Since we are performing a reduction, PROC! must designate a normal-form function designator, namely a *closure*. (Later we will see just how LAMBDA constructs closures are constructed.) Closures contain environments designators, patterns, and bodies, which may be accessed with the selector functions ENVIRONMENT, PATTERN, and BODY, respectively. The result of the reduction is, then, just the result of *expanding the closure* (i.e., normalizing the body of the closure in the environment produced by augmenting the environment captured in the closure with new variable binding obtained by matching the closure's parameter pattern against the normalized argument structure). This is the prescription to be followed for simple 3-LISP procedures.

The second case, the one useful only for primitive procedures, is as follows:

```
(define REDUCE-PRIMITIVE-SIMPLES          ; Demonstration model — not for actual use.
   (lambda simple [proc args env]
      (let [[proc! (normalise proc env)]
            [args! (normalise args env)]]
        ↑(↓proc! . ↓args!))))
```

Here we see a much less elucidating explanation of how a reduction is done. In effect, it says "normalize PROC and ARGS, then just shift levels and go ahead and do it!". It turns out that this game *must* be played for the primitives because there isn't a more-detailed explanation of how a primitive is carried out (at least, not from within 3-LISP; if you are unconvinced, try writing a definition for the standard procedure CAR using only the 3-LISP standard procedures).

Combining these two cases, we come up with a (non-CPS) version of REDUCE that will handle all reductions involving simple procedures:

```
(define REDUCE-SIMPLES                  ; Demonstration model — not for actual use.
   (lambda simple [proc args env]
      (let [[proc! (normalize proc env)]
            [args! (normalize args env)]]
         (if (primitive proc!)
             ↑(↓proc! . ↓args!)
             (normalize (body proc!)
                        (bind (pattern proc!) args! (environment proc!)))))))
```

Its CPS counterpart is as follows:

```
(define REDUCE-SIMPLES                  ; Demonstration model — not for actual use.
   (lambda simple [proc args env cont]
      (normalize proc env
         (lambda simple [proc!]
            (normalize args env
               (lambda simple [args!]
                  (if (primitive proc!)
                      (cont ↑(↓proc! . ↓args!))
                      (normalise (body proc!)
                                 (bind (pattern proc!) args! (environment proc!))
                                 cont)))))))))
```

This brings us to the treatment of reflective procedures, which up to this point have not been explained for reasons that will soon become apparent. In stark contrast to simple procedures, which are run *by the reflective processor*, a **reflective procedure** is one that is run *at the same level as the reflective processor.* Reflective procedures are always passed exactly three arguments: an un-normalized argument structure, the current environment, and the current continuation. A reflective procedure is completely responsible for the remainder of the reduction process for that redex. Here is a overly-simplified version of REDUCE that illustrates how reflective procedures are handled:

```
(define REDUCE-REFLECTIVES              ; Demonstration model — not for actual use.
   (lambda simple [proc args env cont]
      (normalize proc env
         (lambda simple [proc!]
            (↓(de-reflect proc!) args env cont)))))
```

Here we see that the structure that PROC! designates is converted (in an as yet unexplained manner) to a procedure that is then just *called* from the reflective processor with the entire state of the computation (i.e., the environment and continuation). What you are seeing here is one of the essential aspects of reflection: a piece of object-level (user) code is run as part of the reflective processor that is at that very instant running his program. (This is the hook to end all hooks!) In a moment we will demonstrate the elegant power of reflective procedures; for the time being, let's complete our presentation of REDUCE. In 3-LISP, all closures have a procedure-type field that indicates whether it is a simple or a reflective procedure; the utility procedure REFLECTIVE is used to recognize reflective closures; DE-REFLECT converts a reflective closure into a simple one. Integrating the last two CPS versions of REDUCE nets us the version that is actually used in the current 3-LISP reflective processor (again):

```
(define REDUCE
   (lambda simple [proc args env cont]
      (normalize proc env
         (lambda simple [proc!]                              ; Continuation C-PROC!
            (if (reflective proc!)
               (↓(de-reflect proc!) args env cont)
               (normalize args env
                  (lambda simple [args!]                     ; Continuation C-ARGS!
                     (if (primitive proc!)
                        (cont ↑(↓proc! . ↓args!))
                        (normalize (body proc!)
                           (bind (pattern proc!) args! (environment proc!))
                           cont)))))))))
```

Two more standard continuations (again, continuation *families*), called C-PROC! and C-ARGS!, correspond to intermediate steps in the normalization of a pair. C-PROC! accepts the normalized procedure and either passed the buck to a reflective procedure, or initiates the normalization of argument structure. C-ARGS! accepts the normalized argument structure and is responsible for selecting the appropriate treatment for the simple closure, based on whether or not it is recognized as one of the primitive closures.

2.b.vi. READ-NORMALIZE-PRINT

There is one other part of the 3-LISP system to be explained: READ-NORMALIZE-PRINT, 3-LISP's top-level driver loop. This is the behavior one might expect from it:

```
1> (READ-NORMALIZE-PRINT 99 GLOBAL PRIMARY-STREAM)
99> (+ 2 2)
99= 4
99>
```

In other words, READ-NORMALIZE-PRINT is responsible for cycling through the issuing of a prompt, the reading of the user's input expression, the normalizing of it, and the subsequent displaying of the result. Here is how it is defined:

```
(define READ-NORMALIZE-PRINT
   (lambda simple [level env stream]
      (normalize (prompt&read level stream) env
         (lambda simple [result]                    ; Continuation C-REPLY
            (block (prompt&reply result level stream)
                   (read-normalize-print level env stream))))))
```

Which brings us to the important question of just how is the system initialized. Recall that in a reflective model, object-level programs are run by the reflective processor one level up; in turn, this reflective processor is run by another instance of the reflective processor one level above it; and so on, *ad infinitum*. In 3-LISP, each reflective level of the processor is assumed to start off running READ-NORMALIZE-PRINT. The way this is imagined to work is as follows: the very top processor level (infinitely high up) is invoked by someone (say, God, or some functional equivalent) normalizing the expression '(READ-NORMALIZE-PRINT ∞ GLOBAL PRIMARY-STREAM)'. When it reads an expression, it is given an input string requesting that a new top-level, numbered one lower, should be started up; and so forth, until finally the second reflective level is given '(READ-NORMALIZE-PRINT 1 GLOBAL PRIMARY-STREAM)'. This types out '1>' on the console, and awaits *your* input. I.e., if it hadn't scrolled off your screen, you'd have seen the genesis transcript that goes as follows:

```
god> (READ-NORMALIZE-PRINT  ∞  GLOBAL PRIMARY-STREAM)
∞>  (READ-NORMALIZE-PRINT ∞-1 GLOBAL PRIMARY-STREAM)
∞-1>(READ-NORMALIZE-PRINT ∞-2 GLOBAL PRIMARY-STREAM)
        •
        •
        •
3> (READ-NORMALISE-PRINT 2 GLOBAL PRIMARY-STREAM)
2> (READ-NORMALISE-PRINT 1 GLOBAL PRIMARY-STREAM)
————————— You came along here —————————
1>
```

The initialization sequence is another essential part of a reflective system, since it determines the initial state (i.e., environment and continuation) at each reflective level. One usually becomes aware of these matters when one starts writing reflective procedures that break the computational chain letter, so to speak, by neglecting to call their continuation (it is for exactly this eventuality that each reflective level identifies itself with its own distinctive prompt).

```
1> (define FORGETFUL
       (lambda reflect [[] env cont]
          'SIGH!))
1= 'FORGETFUL
1> (FORGETFUL)
2= 'SIGH!
2> (FORGETFUL)
3= 'SIGH!
```

This completes the description of the core of the reflective processor, READ-NORMALIZE-PRINT (with its continuation, C-REPLY) and the so-called "magnificent seven" mutually-recursive *primary processor procedures* (*ppp*'s): three named procedures (NORMALIZE, REDUCE, and NORMALIZE-RAIL) and four standard continuations (C-PROC!, C-ARGS!, C-FIRST!, and C-REST!).

2.b.vii. Reflective Procedures

As promised earlier, we are now in a position to show how reflective procedures can be put to use. Just remember that when a reflective procedure is called, the body of it gets run at the level of the reflective processor one level up. A reflective procedure can cause the processing in progress to proceed with a particular result simply by calling the continuation with the desired structure. The following silly example illustrates a reflective procedure appropriately called THREE that behaves exactly like the constant function of no arguments that always has the value three.

```
1> (define THREE
       (lambda reflect [[] env cont]
          (cont '3)))
1= 'THREE
1> (THREE)
1= 3
1> (+ 100 (THREE))
1= 103
1> (+ (THREE) (THREE))
1= 6
```

On the other hand, a reflective procedure may request that an expression be normalized by explicitly calling NORMALIZE (or REDUCE, if appropriate), as the following version of ID (the identity function) demonstrates:

```
1> (define NEW-ID
       (lambda reflect [[exp] env cont]
          (normalize exp env cont)))
1= 'NEW-ID
1> (NEW-ID (+ 2 2))
1= 4
1> (+ 100 (NEW-ID (+ 2 2)))
1= 104
```

Before moving on to some justifiable uses of reflective procedures, we just can't resist the urge to write the old hackneyed factorial procedure as a lambda-reflect:

```
1> (define REFLECTIVE-FACTORIAL
      (lambda reflect [[exp] env cont]
         (normalize exp env
           (lambda simple [exp!]
              (if (= exp! '0)
                  (cont '1)
                  (cont ↑(* ↓exp! (reflective-factorial (1- ↓exp!))))))))
1= 'REFLECTIVE-FACTORIAL
1> (REFLECTIVE-FACTORIAL (+ 2 2))
1= 24
1> (+ 100 (REFLECTIVE-FACTORIAL 5))
1= 220
```

Okay! Okay! We'll confine our attention to situations where reflective procedures are really necessary. Simple procedures turn out to be inadequate for defining control operators for a number of reasons. Examples where reflective procedures are needed: IF, where some of the arguments may not be normalized; LAMBDA and SET, where explicit access to the current environment is required; and CATCH, where explicit access to the current continuation is required. We will consider each of these, in turn, beginning with IF. (Note that the actual- i.e., Appendix A- definitions of these control operators differ in several rather uninteresting ways from the ones we will present here.)

```
1> (define NEW-IF
      (lambda reflect [[premise consequent antecedent] env cont]
         (normalize premise env
            (lambda simple [premise!]
              (if ↓premise!
                  (normalize consequent env cont)
                  (normalize antecedent env cont))))))
1= 'NEW-IF
1> (NEW-IF (= 2 2) (+ 2 2) (error))
1= 4
```

We see that NEW-IF normalizes either its second or its the third argument expression depending on whether the first expression normalized to '$T or '$F, respectively. Moreover, all normalizations are done in the current environment. Notice that the above definition of NEW-IF makes use of IF — which seems like a cheap trick. The following definition of NEWER-IF makes use of the primitive (and therefore simple) procedure EF in conjunction with an idiomatic use of LAMBDA known as ***λ-deferral***.

```
1> (define NEWER-IF
      (lambda reflect [[premise consequent antecedent] env cont]
         (normalize premise env
            (lambda simple [premise!]
              ((ef ↓premise!
                  (lambda simple [] (normalize consequent env cont))
                  (lambda simple [] (normalize antecedent env cont))))))))
1= 'NEWER-IF
1> (NEWER-IF (= 2 2) (+ 2 2) (error))
1= 4
```

Next we look at SET (Note: That's 3-LISP's assignment statement, known in most other LISP dialects as SETQ.). Besides the desire to avoid normalizing the first argument of a SET redex (the variable), explicit access to the current environment will be required to complete the processing. (REBIND does the actual work of modifying the environment designator.)

```
1> (define NEW-SET
      (lambda reflect [[var exp] env cont]
         (normalize exp env
           (lambda simple [exp!]
              (block
                 (rebind var exp! env)
                 (cont ''ok))))))
1= 'NEW-SET
1> (NEW-SET BLEBBIE (+ 100 100))
1= 'OK
1> BLEBBIE
1= 200
```

We will now show how LAMBDA can be defined in stages, beginning with a stripped-down version LAMBDA-SIMPLE:

```
1> (define LAMBDA-SIMPLE
      (lambda reflect [[pattern body] env cont]
         (cont (ccons 'simple ↑env pattern body))))
1= 'LAMBDA-SIMPLE
1> (LAMBDA-SIMPLE [X] (* X X))
1= {closure}
1> ((LAMBDA-SIMPLE [X] (* X X)) 10)
1= 100
1> (TYPE (LAMBDA-SIMPLE [X] (* X X)))
1= 'FUNCTION
```

LAMBDA-SIMPLE simply constructs a new closure containing an indication that it is a simple closure, the current environment (or rather, designator thereof), and the pattern and body structures exactly as they appeared in the LAMBDA-SIMPLE redex. LAMBDA-REFLECT differs from LAMBDA-SIMPLE only in the choice of atom used in the procedure-type field of the closure.

```
1> (define LAMBDA-REFLECT
      (lambda reflect [[pattern body] env cont]
         (cont (ccons 'reflect ↑env pattern body))))
1= 'LAMBDA-REFLECT
1> ((LAMBDA-REFLECT [ARGS ENV CONT] (CONT ''?)) (error))
1= '?
```

In the interest of being able to define not only simple and reflective procedures, we can devise a general λ-abstraction operator that takes, as its first argument, an expression designating a function to be used to do the work. This function applied to three arguments — the designator of the current environment, the pattern structure, and the body structure — designates a new function.

```
1> (define NEW-LAMBDA
      (lambda reflect [[kind pattern body] env cont]
         (reduce kind ↑[↑env pattern body] env cont)))
1= 'NEW-LAMBDA

1> (define NEW-SIMPLE
      (lambda simple [def-env pattern body]
         ↓(ccons 'simple def-env pattern body)))
1= 'NEW-SIMPLE

1> (define NEW-REFLECT
      (lambda simple [def-env pattern body]
         ↓(ccons 'reflect def-env pattern body)))
1= 'NEW-REFLECT
```

```
1> (NEW-LAMBDA NEW-SIMPLE [X] (* X X))
1= {closure}
1> ((NEW-LAMBDA NEW-SIMPLE [X] (* X X)) 10)
1= 100
1> (TYPE (NEW-LAMBDA NEW-REFLECT [ARGS ENV CONT] (CONT ''?)))
1= 'FUNCTION
```

With this general abstraction mechanism in place, it is a simple thing to define ***macros***. These are procedures that are reduced by first constructing a different structure out of the argument expressions, and then normalizing this structure in place of the original redex. The body of the macro procedure describes how to do the expansion; i.e., it maps structures into other structures. For example, we can define a macro procedure BUMP so that any redex of the form (BUMP *VAR*) will be converted into one of the form (SET *VAR* (1+ *VAR*)).

```
1> (define NEW-MACRO
       (lambda simple [def-env pattern body]
          (let [[expander (SIMPLE def-env pattern body)]]
             (lambda reflect [args env cont]
                (normalize (expander . args) env cont)))))
1= 'NEW-MACRO

1> (define BUMP
       (lambda NEW-MACRO [var]
          (xcons 'set var (xcons '1+ var))))
1= 'BUMP
1> (SET BUMPUS 1)
1= 'OK
1> (BUMP BUMPUS)
1= 'OK
1> BUMPUS
1= 2
```

The back-quote feature (see §3.b.) is very useful when it comes to defining the bodies of macro procedures. For example, LET is defined as a macro utilizing back-quote, based on the following transformation:

$$(\text{LET } [[V_1\ E_1][V_2\ E_2]\ \ldots\ [V_n\ E_n]]\ BODY)$$

expands to

$$((\text{LAMBDA SIMPLE } [V_1\ V_2\ \ldots\ V_n]\ BODY)\ E_1\ E_2\ \ldots\ E_n)$$

```
1> (define NEW-LET
       (lambda new-macro [list body]
          `((lambda simple ,(map 1st list) ,body) . ,(map 2nd list))))
1= 'NEW-LET
1> (NEW-LET [[X 1]] (+ X 2))
1= 3
```

As a final example of the power of reflective procedures, we shall define SCHEME's CATCH operator:

```
1> (define SCHEME-CATCH
       (lambda reflect [[catch-tag catch-body] catch-env catch-cont]
          (normalize catch-body
             (bind catch-tag
                   ↑(lambda reflect [[throw-exp] throw-env throw-cont]
                       (normalize throw-exp throw-env catch-cont))
                   catch-env)
             catch-cont)))
1= 'SCHEME-CATCH
```

```
1> (+ 2 (+ 5 10))
1= 17
1> (+ 2 (SCHEME-CATCH ESCAPE (+ 5 10)))
1= 17
1> (+ 2 (SCHEME-CATCH ESCAPE (ESCAPE (+ 5 10))))
1= 12
1> (+ 2 (SCHEME-CATCH ESCAPE (+ 5 (ESCAPE 10))))
1= 12
1> (+ 2 (SCHEME-CATCH ESCAPE
          (BLOCK (ESCAPE 10)
                 (PRINT 'GOTCHA PRIMARY-STREAM))))
1= 12
```

2.b.viii. Reflective Protocols

Unless you have a particular reason to do otherwise, the following protocols concerning reflective programming should be kept in mind:

★ CPS procedures (this includes reflecive procedures) should always call continuations and other CPS procedures from a tail-recursive position. That way, the explict continuation will always reflect the remainder of the computation.

★ CPS procedures should either call their continuation or pass it along to another CPS procedure.

★ Continuations should be called with a single structure-designating argument.

2.b.ix. A Note on Recursion and the Y-Operator

Closures created via the standard procedure DEFINE capture the current environment augmented by the binding of the procedure variable to the designator of the closure. This circularity is created via Y-OPERATOR, a variation on Church's paradoxical combinator. (For further explanation, see 4.c.8. of Smith's thesis.)

3. 3-LISP Structures and Notation

3-LISP is based on a serial model of computation, consisting of a topology or graph of structures collectively called a ***structural field***, examined and manipulated by a single active ***processor***. This section describes the elements of 3-LISP's structural field, and the notation used to display them.

3.a. Structural Field

Objects in the structural field are called ***internal structures*** or, when it is not confusing, just ***structures;*** mathematical entities like numbers and sequences are called ***external structures*** or ***abstractions*** (but never just *structures*). The ***world*** consists of entities of all sorts, including both internal and external structures, and undoubtedly many other things as well.

There are exactly nine ***types*** (kinds, categories) of internal structures that populate the structural field. This immutable property of each structure may be interrogated with the standard procedure TYPE. The standard procedure = can be used to test to see if they are one and the same structure.

Type	*Designation*	*Normal*	*Constructor*	*Standard Notation*
Numerals	Numbers	Yes	—	*a sequence of digits*
Booleans	Truth-Values	Yes	—	$T *or* $F
Charats	Characters	Yes	—	*#character*
Streamers	Streams	Yes	—	{streamer}
Closures	Functions	Yes	CCONS	{closure}
Atoms	(Designation of Binding)	No	ACONS	*a sequence of alphanumerics*
Pairs	(Value of Application)	No	PCONS	(*EXP* . *EXP*)
Rails	Sequences	Some	RCONS	[*EXP EXP* ... *EXP*]
Handles	Internal Structures	Yes	—	'*EXP*

Recall that a structure is said to be in *normal form* if it cannot be further simplified by the processor. A normal-form structure S_1 is ***canonical*** if all co-designating structures, S_2, normalize to S_1. Note that six- and- a- half of the categories are normal-form structures, and that all five of the non-constructible (i.e., permanent) structure types are canonical.

Each of these nine structure types can be briefly described:

Numerals: There are an infinite number of 3-LISP integer numerals, set in one-to-one correspondence to the abstract external numbers (ultimately we intend to support full rational or repeating fraction arithmetic, but at the moment only integers are defined). All numerals are canonical normal-form designators of numbers.

Booleans: There are just two boolean structures, notated as '$T' and '$F', that are constants (rigid designators) of Truth and Falsity, respectively. These normal-form structures may be viewed as the canonical true and false statements.

Charats: We do not claim to know what characters are, but charats are their normal-form designators. More precisely, a charat is an atomic structure associated one-to-one with character *types* (in the linguist's sense); there is only one charat for the character '+', although there, of course, may be an arbitrary number of tokens (occurrences) of that character.

Streamers: Streams are intended to serve as the interface with the outside world (i.e., to function essentially as communication channels); as a consequence, we say virtually nothing about them, other than that they are legimitate arguments for INPUT and OUTPUT. Streamers are their normal-form designators. Note that no field relationships are defined over streamers. Streamers will probably play a role in implementations and embeddings of 3-LISP, but at present the language puts no specific constraints on the way in which this role is played.

Closures: Closures are normal-form function designators; because we have no adequate theory of procedural intension, they retain *all* the relevant contextual information from the point of function definition (expression and enclosing environment). Although closures, being first-class structures, can be inspected and compared, closure identity is far more fine grained than function identity.

Atoms: As in standard LISPs, atoms are atomic structures used as variables (schematic names). Atoms are associated with identifiers (lexical spellings) only through the READ and PRINT functions.

Pairs: Pairs are exactly as in LISP 1.5: they are ordered pairs, consisting of a CAR and a CDR (which may in turn be any structure in the field). Unlike standard LISPs, however, 3-LISP pairs are used for only one purpose: to encode function applications (a pair is therefore sometimes called a ***redex***, for 'reducible expression').

Rails: Rails, derivative from standard LISP's lists, are used to designate abstract sequences. Like the lists of LISP 1.5, isomorphic rails may be distinct. Those rails whose elements are normal-form are, by definition, themselves in normal-form; thus the rail `[1 2]` is in normal-form, whereas the rail `[1 (+ 1 1)]` is not.

Handles: Handles are unique normal-form designators of other internals structures — they are the 3-LISP field's form of canonical quotation. Thus for the atom `X` there is a single handle, written `'X`. All 3-LISP structures have handles (including handles themselves; thus the handle of the handle of the atom `X` is `''X`).

The nine first-order locality relationships defined over internal structures are summarized in the following table:

Name	*Type*	*Total*	*→ Accessible*	*←*	*Standard Procedure*
CAR	Pairs → Structures	Yes	Yes	No	CAR
CDR	Pairs → Structures	Yes	Yes	No	CDR
FIRST	Rails → Structures	No	Yes	No	1ST
REST	Rails → Rails	No	Yes	No	REST
PROC-TYPE	Closures → Atoms	Yes	Yes	No	PROCEDURE-TYPE
ENV	Closures → Rails	Yes	Yes	No	ENVIRONMENT-DESIGNATOR
PATTERN	Closures → Structures	Yes	Yes	No	PATTERN
BODY	Closures → Structures	Yes	Yes	No	BODY
REF	Handles → Structures	Yes	Yes	Yes	DOWN (↓)

All of these relations are, in fact, total functions, with the exception of FIRST and REST, which are only partial, being undefined together on empty rails. REF is one-to-one and onto; therefore REF^{-1} is a total function on structures, called HANDLE, and is a subset of the function that is designated by the standard procedure UP (↑).

Some structures — all numerals, charats, booleans, streamers, and their handles — are permanent members of any structural field configuration. Others — pairs, rails, atoms, and closures — can be brought into existence and connected to existing structures through the activation of one of the primitive constructors. For example, the standard procedures called PCONS creates a new pair and establishes a CAR and CDR relationship between this pair and the two structures passed to PCONS as arguments.

A structure X is ***accessible*** from structure Y if X can be reached from Y through a series of CAR, CDR, FIRST, etc., connections. In addition, the handles of all structures are accessible from their referents. When a so-called 'new' structure is generated (by RCONS, PREP, SCONS, ACONS, CCONS, or PCONS) it is guaranteed to be *otherwise* ***inaccessible***, meaning that it cannot be accessed from any other accessible structure. A rail is considered to be ***completely inaccessible*** if it and all of its tails (i.e., rails reachable via one or more REST transitions) are inaccessible. Thus RCONS returns an otherwise completely inaccessible rail, whereas PREP returns an inaccessible, but not completely inaccessible rail.

Once created, a structure will remain a part of the structural field permanently, unless it is smashed by REPLACE, the primitive ***structural field side-effect*** procedure. Replacing structure S_1 by S_2 has the effect of permanently altering the topology of the structural field such that all structures that were mapped to S_1 via one of the nine locality functions become mapped to S_2. As a result, S_1 and all its handles suddenly become completely inaccessible. Both S_1 and S_2 must be of the same type, and that type must be one of the non-canonical ones: rail, pair, closure, or atom.

3.b. Standard Notation

The 3-LISP ***internalization function*** (the notational interpretation function O that maps notations into internal structures) is not, strictly speaking, a primitive part of the language definition, since it is not used in internal processing (i.e., discarding it will not topple the tower). There is, however, what is called a ***standard notation*** that is used in all documentation (including this reference guide), and which is provided with a 3-LISP system upon initialization. (A user may, however, completely replace it with his/her own version, if desired). This section explains that notation.

The lexical notation is designed to satisfy three goals:

1. In so far as possible, to resemble standard LISP notational practice;
2. To maintain category alignment with the field (one lexical type per structural type);
3. To be convenient.

The goal of category alignment is met by having the standard notation for each type be identifiable in the first character (except for "notational escapes," described below), as indicated in the following chart:

Type	*Leading Character*	*Examples*
Numeral	*digit*	`0, 1, -24, +100, 007`
Atom	*letter*	`A, REDUCE, CAR, ATOM`

Boolean	`$`	`$T, $F`
Pair	`(`	`(A . B), (PLUS 2 3)`
Rail	`[`	`[1 2 3], []`
Handle	`'`	`'A, '(+ 2 3), ''[]`
Charat	`#`	`#A, #/`
Other	`{`	`{closure}, {streamer}`

Examples: '`(A . 1)`' notates a pair whose CAR is the atom notated '`A`' and whose CDR is the numeral notated '`1`'; '`[]`' notates an empty rail; '`[1 2 3 4 5 6]`' notates a rail whose FIRST and REST would be notated '`1`' and '`[2 3 4 5 6]`', respectively; '`'100`' notates the handle for the numeral '`100`'.

Some subtleties complicate this clean correspondence. Specifically:

1. Numerals can have a leading '`+`' or '`-`' (i.e., '`-24`').
2. An atom label may begin with a digit (or sign) providing it contains at least one non-digit (i.e., '`6N237E`', '`-X`' and '`1+`' are valid atom labels). Any atom label that also satisfies the rules for numeral tokens will be taken to be the latter. For example, '`1-`' and '`+`' notate atoms, whereas '`+1`' notates a numeral.
3. Left brace ('`{`') is used as a *general* notational escape, not only for closures and streamers, but also for unlabelled atoms, errors, and other notational commentary. This notation is currently employed only on *output.*
4. Case is ignored in atom labels (converted to upper case on input). For example, '`Zaphod`', '`ZAPHOD`', '`zaphod`', and '`ZaPhOd`' all notate the same atom.
5. Some lexical abbreviations (notational sugar) are supported:

'`(`exp_1 exp_2 ... exp_k`)`'	*abbreviates*	'`(`exp_1 `. [`exp_2 ... exp_k`])`'
'"$char_1$ $char_2$ ... $char_k$"'	*abbreviates*	'`'[#`$char_1$ `#`$char_2$ ... `#`$char_k$`]`'
'↑*exp*'	*abbreviates*	'`'(UP` *exp*`)`'
'↓*exp*"	*abbreviates*	'`'(DOWN` *exp*`)`'

The following grammar presents the essence of the standard 3-LISP notation, for those who like such things:

Extended BNF Grammar for 3-LISP Standard Notation

1.	Expression	::=	Regular | Abbreviation | Escape
2.	Regular	::=	Numeral | Boolean | Charat | Atom | Pair | Rail | Handle
	Numeral	::=	[Sign-character] Digit-character$^+$
	Boolean	::=	'`$T`' \| '`$F`' | '`$t`' \| '`$f`'
	Charat	::=	'`#`' Any-character
	Atom	::=	Atom-character$^+$
	Pair	::=	'`(`' Expression '`.`' Expression '`)`'
	Rail	::=	'`[`' Expression* '`]`'
	Handle	::=	'`'`' Expression
3.	Abbreviation	::=	Up | Down | Extended-pair | String | Back-Quote | Comma
	Up	::=	'↑' Expression
	Down	::=	'↓' Expression

	Extended-pair	::=	'(' Expression⁺ ')'
	String	::=	'"' (String-character | '"' '"')* '"'
	Back-Quote	::=	'`' Expression
	Comma	::=	',' Expression
4.	Escape	::=	'{' *Unspecified information* '}'
5.	Sign-character	::=	'+' | '-'
	Digit-character	::=	'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
	Any-character	::=	*Any character in the character set*
	Atom-character	::=	*Any character except Space, End-of-line,* '$', '#', '"', ';', or *Special*
	String-character	::=	*Any character except* '"'
	Normal-character	::=	*Any character except End-of-line*
	Special	::=	'(' | ')' | '[' | ']' | '↑' | '↓' | '.' | ',' | '{' | '}' | ''' | '`'
6.	Token-sequence	::=	(Separator* Token Separator*)*
	Separator	::=	Space-character | End-of-line-character | Comment
	Comment	::=	';' Normal-character* End-of-line-character

In the standard notation, structures are notated with sequences of ***lexical tokens***, each of which is composed of a sequence of one or more characters chosen from a collection of characters called the ***character set***. Although the exact composition of the character set is unimportant, we assume that it includes all of the ASCII characters.

Sequences of characters are broken down into tokens in the conventional way, with the rule that there must always be at least one token separator between adjacent non-special tokens. For example, the character stream '(foo [1 $T ↑#x ''100])' consists of the ten tokens: '(', 'foo', '[', '1', '$T', '↑', '#x', '''100' ']', and ')'.

Special tokens do not notate structures by themselves; rather, they are used to punctuate the notation for composite structures.

For convenience, the following table lists all "special" (i.e., non alpha-numeric) characters that are used for some special purpose. Note that the standard notation uses one character (down-arrow: '↓') that is not part of the standard ASCII character set, but we reserve ASCII backslash ('\') so that it can be used in its stead. We assume, in other words, that the 3-LISP standard notation is indeed based on the standard ASCII sequence, but simply choose to *print* backslash as a down-arrow.

Code, Character, and Use		*Code, Character, and Use*	
(	— starts pairs	↑	— abbreviation for UP
)	— ends pairs	↓	— abbreviation for DOWN (same as '\')
.	— separates CAR and CDR	`	— back-quote
[	— starts rails	,	— normalized expression within back-quote
]	— ends rails	;	— starts comments (to CRLF)
'	— handles	-	— starts negative numerals
#	— charats	+	— starts some positive numerals
$	— booleans ($T and $F)	{	— starts notational escapes
"	— starts and ends strings	}	— ends notational escapes

In addition to the foregoing notational protocols embodied in the internalizer and externalizer, we adopt a set of additional notational *conventions* on identifiers. The 3-LISP system pays no attention to these, but all of the code presented in this manual honors them, and they are recommended for users, as well. Specifically:

1. A suffix exclamation point is used on variables (atoms) that are intended always to designate a normal-form structure. For example:

   ```
   (NORMALIZE (REST EXP) ENV (LAMBDA SIMPLE [REST!] (CONT (PREP FIRST! REST!)))).
   ```

2. A suffix asterisk is used on variants of procedures that take an indefinite number of arguments, where the standard version accepts only a fixed number (for example: ID* is a multi-argument version of ID).

A general comment: The internalization function described above is not onto; in other words, there are structures that are not the result of internalizing any lexical expression, for two reasons. First, upon internalization, all pairs and rails notated are created from previously-inaccessible cells. Hence, any structure with a shared sub-structure will have no lexical counterpart. Second, closures and un-named atoms (those created by ACONS) have no standard lexical counterparts. The standard version of PRINT, approximately the inverse of READ, currently makes no attempt to deal in a sophisticated way with either of these problems. In particular, no attempt is made to show shared substructure, and un-notatable structures — closures, nameless atoms, and circular structures — are marked with a standard lexical escape: a note enclosed in braces (e.g., '{closure}', '{streamer}', etc.).

The back-quote feature, borrowed from MACLISP [Moon 74], is useful when defining macros, since it allows one to conveniently notate expressions for constructing structures that resemble the lexical expression notated. For example, '`(A . B)' is notationally equivalent to '(PCONS 'A 'B)', which notates a structure that normalizes to a structure that would be notated ''(A . B)'. The comma notation, meaningful only within the scope of a back-quote, gives one a fill-in-the-blanks-like capability; for example, the notation '`(A . ,X)' is shorthand for '(PCONS 'A X)', which notates a structure that would normalize to a structure that would be notated '(A . HELLO)' in an environment where X was bound to 'HELLO. The following examples may help to make the workings of this feature clear (assume that B is bound to '2 and C to ''3):

Notation	*abbreviates*	*and normalizes to*
`1	'1	'1
`(A . B)	(PCONS 'A 'B)	'(A . B)
`[A B]	(RCONS 'A 'B)	'[A B]
`[A ,B C]	(RCONS 'A B 'C)	'[A 2 C]
`[[A B][,C D]]	(RCONS '[A B] (RCONS C 'D))	'[[A B]['3 D]]
``[A ,B ,,C]	(PCONS ('RCONS `['A B ,C]))	'(RCONS ['A B '3]) (i.e., `[A ,B 3])

To express the workings of this mechanism precisely requires a little care, since both notation and designation must be spoken of explicitly. It can be summarized as follows:

> ***Back-Quote Principle:*** *A lexical expression E_1 preceded by a back-quote will notate a structure S_1 that designates a structure S_2 that would be notated by E_1, with the exception that those fragments of S_2 that would be notated by portions of E_1 that are preceded by a comma will, in fact, be designated by the structures that those portions notate, rather than notated by them directly.*

An intensional note: the back-quote expander will not use a token *tail* of a rail if any part of that rail has a comma'ed expression within it. Specifically, we have:

```
1> (DEFINE TEST
      (LAMBDA SIMPLE [A]
         `[,A 2 3]))
=> TEST
1> (LET [[X (TEST '1)]
         [Y (TEST '2)]]
      (= (REST X) (REST Y)))
=> $F
```

since `` `[,A 2 3] `` abbreviates `(RCONS A '2 '3)`, not `(PREP A '[2 3])`.

See the Appendix A definitions of DEFINE, LET, LETSEQ, and other macros for further examples.

4. Standard Procedures

There are approximately 150 ***standard procedures*** in 3-LISP: procedures that are described in this reference guide, used without comment in utility packages, and so forth (we also expect to 'compile' these procedures into the standard implementation). A 3-LISP programmer should consider these to be the base set, on top of which to define other functionality as desired. Within the set of standard procedures, however, are two important sub-classes: *primitive* procedures that provide access to the structural field and to the external world (e.g., I/O); and *kernel* procedures that are essential to the workings of the system. These two sets are neither mutually exclusive nor exhaustive: many of the primitives are kernel procedures as well (EMPTY, for example), but there are some non-kernel primitives (LENGTH, +, ACONS, REPLACE, etc.). In addition, it is clear that many kernel procedures are not primitive (LAMBDA, BINDING, NORMALISE, and NORMAL, to name a few). Finally, there are approximately 90 other standard procedures (MAX, LABELS, DO, etc.) that are neither primitive nor kernel.

4.a. Primitive Procedures

There are 34 ***primitive procedures*** (listed below) that have no definition within 3-LISP, and that are reduced with arguments in "unit time," in the sense that from no level of reflective access is there any visible grain to their operation. All the 3-LISP primitives are simple: there are no primitive reflectives. To a certain extent the particular set is arbitrary, and it is certainly not minimal: SCONS, for example, could be defined in terms of RCONS, UP, and DOWN; LENGTH could be defined in terms of EMPTY and +; etc.

Category	*Standard Name*	*Functionality*
Typing:	TYPE	defined on 15 types (9 internal, 6 external)
	PROCEDURE-TYPE	to distinguish simple and reflective closures
Identity:	=	defined on 14 types (all except functions)
Structural:	PCONS, CAR, CDR	to construct and examine pairs
	CCONS, PATTERN, BODY	to construct and
	ENVIRONMENT-DESIGNATOR	examine closures
	ACONS	to construct atoms
	RCONS, SCONS, PREP	to construct and examine
	LENGTH, NTH, TAIL, EMPTY	rails and sequences
Modifier:	REPLACE	to modify mutable structures
Control:	EF	an extensional if-then-else conditional
Semantics:	UP, DOWN	to mediate between sign & signified
Arithmetic:	+, -, *, /, <, >,<=,>=	as one would expect
I/O:	INPUT, OUTPUT	primitive operations on streams
System:	LOADFILE, EDITDEF	system support

4.b. Kernel Procedures

The ***kernel procedures*** are those that are *used crucially in the* 3-LISP *reflective processor* (i.e., they are used by the reflective processor to process the reflective processor). As a consequence, smashing one of these closures, or redefining the binding of its standard name in the global environment (more accurately: in any environment captured inside any of the kernel closures), will cause the tower to fall. Thus, for all practical purposes, the kernel procedures are as 'wired-in' to 3-LISP as are the primitives, even though in a strict sense they have visible definitions, and are compositionally executed by the processor (by expanding closures). Note that there are reflective kernel procedures as well as simple ones. It turns out that the kernel procedures are exactly the acquaintances of NORMALIZE, although this needn't have been so (they could have been a subset, since there might have been code in the reflective processor that, although used when processing some forms of user code, didn't happen to be used to process the processor itself).

Kernel Primitives

CAR, CDR, RCONS, SCONS, PREP, NTH, TAIL, EMPTY, CCONS, PROCEDURE-TYPE, ENVIRONMENT-DESIGNATOR, PATTERN, BODY, TYPE, =, EF, UP, DOWN

Kernel Non-primitives

UNIT, DOUBLE, REST, 1ST, 2ND, MEMBER, VECTOR-CONSTRUCTOR, MAP, ENVIRONMENT, REFLECTIVE, DE-REFLECT, ATOM, PAIR, RAIL, HANDLE, EXTERNAL, LAMBDA, SIMPLE, BINDING, BIND, LET, IF, COND, COND-HELPER, AND, AND-HELPER, NORMALISE, REDUCE, NORMALISE-RAIL, NORMAL, NORMAL-RAIL, PRIMITIVE

4.c. Standard Procedure Guide

The remainder of this section is taken up with descriptions of each of the standard procedures. The 3-LISP code for the standard procedures can be found in Appendix A. Notes on the format of these descriptions:

1. Each procedure is illustrated with non-objectified arguments, but many can be used in other ways (for example: (PCONS . (REST ['A 'B 'C])) ⇒ '(B . C)).
2. For each procedure, we give the declarative import. In many cases that is the only semantical information provided, since if the designation has a canonical normal-form designator, what is *returned* can be determined from this designation in conjunction with the normal-form theorem. For example, since (+ 2 3) designates the number 5, it will return the numeral 5; since (= 'A 'B) designates falsity, it will return the boolean $F. If, however, the normal-form designator is not canonical, or if there are side effects, the relevant parts of the procedural significance are described as well.
3. Typing information is typically given only in terms of what we call the functions "Φ-type." Thus, for example, the division function / would be said to have Φ-type of *[NUMBERS × NUMBERS] → NUMBERS*. In some cases, the typing restrictions specified in this section are stricter than one would expect given the Appendix A definitions.
4. Underlined arguments in the title line of a procedure description indicate those positions that are normalized tail recursively with respect to the procedure call (e.g., the 2nd and 3rd arguments to IF).
5. Several one-word attributes are associated with each procedure that can provide a quick reference for determining the nature of the procedure. The following keywords are used:

Cons	This procedure may create new structures that will be accessible from the result; e.g., APPEND.
Smash	Internal structures accessible from the argument designators may be smashed (with REPLACE); e.g., REBIND.

Env	Some of the arguments to this procedure may be normalized in some environment other that the current one; however, these environment manipulations are accomplished through non-destructive means; e.g., LET.
Smash-env	This procedure may destructively change the current environment; e.g., SET.
I/O	This procedure may side effect the outside world by doing I/O; e.g., OUTPUT.
CPS	This procedure is written in the continuation-passing style — instead of returning, the result is explicitly passed to the continuation (usually as the last argument); e.g., NORMALIZE.
Abnormal	Some of the arguments may not always be normalized; e.g., IF.

6. Still other keywords are used to indicate the nature of the procedure's status within the implementation:

Primitive	This procedure is one of the 30 or so primitives that have only viciously circular definitions within the 3-LISP system. All non-primitives have complete and accurate descriptions in terms of the primitives.
Kernel	This procedure is an essential part of 3-LISP because it is used regularly by the reflective processors at all levels.

7. The symbol '⇒' (used in examples) means "normalizes to."
8. Some comments in regard to examples involving I/O: all input expressions are printed in italics following the level 1 processor's '1>' prompt and output expressions appear unitalicized following the '1=' prompt.

```
1> 'HELLO
1= 'HELLO
```

Input destined for an explicit call to READ (or INPUT, etc.) are underlined as well as italicized.

```
1> (READ PRIMARY-STREAM) HELLO
1= 'HELLO
```

Output produced by an explicit call to PRINT (or OUTPUT, etc.) is printed in bold.

```
1> (PRINT 'HELLO PRIMARY-STREAM) HELLO
1= 'OK
```

Note that in the interest of readability several liberties have been taken with the formatting of output expressions — actual results may vary.

9. To facilitate the writing of macros and other reflective procedures, the argument-to-parameter pattern matcher (BIND) will convert a rail-designating argument into a sequence of designators. For example, '[1 2 3] will be converted to ['1 '2 '3] in order to fit the pattern [A B C]. This is consistent with the polymorphism of 1ST and REST, etc. — (1ST '[1 2 3]) and (1ST ['1 '2 '3]) both normalize to '1.
10. All standard procedures return a result. However, the ones that are used solely to accomplish a side-effect (e.g., REPLACE, SET, and OUTPUT) usually return a gratuitous 'OK.

4.c.1. PAIRS

(PCONS S_1 S_2**)**

Designates an otherwise inaccessible pair whose CAR is the internal structure designated by S_1 and whose CDR is the internal structure designated by S_2.

Φ-Type: **[STRUCTURES × STRUCTURES] → PAIRS** *Properties:* Primitive; cons.

Examples:

```
(PCONS 'A 'B)        ⇒  '(A . B)
(PCONS '+ '[2 3])    ⇒  '(+ 2 3)
(PCONS 2 3)          ⇒  {ERROR: Structure expected.}
```

(CAR ***PAIR*****)**

Designates the internal structure that is the CAR of the pair designated by *PAIR*.

Φ-Type: **[PAIRS] → STRUCTURES** *Properties:* Primitive; kernel.

Examples:

```
(CAR '(A . B))    ⇒  'A
(CAR '(1 . $T))   ⇒  '1
(CAR '(+ 2 3))    ⇒  '+
(CAR '+))         ⇒  {ERROR: Pair expected.}
```

(CDR ***PAIR*****)**

Designates the internal structure that is the CDR of the pair designated by *PAIR*.

Φ-Type: **[PAIRS] → STRUCTURES** *Properties:* Primitive; kernel.

Examples:

```
(CDR '(A . B))    ⇒  'B
(CDR '(1 . $T))   ⇒  '$T
(CDR '(+ 2 3))    ⇒  '[2 3]
(CDR '(ACONS))    ⇒  '[]
(CDR '1))         ⇒  {ERROR: Pair expected.}
```

(XCONS S_1 S_2 ... S_k**)**

Designates an otherwise inaccessible pair whose CAR is the internal structure designated by S_1 and whose CDR is an otherwise completely inaccessible rail whose elements are the internal structures designated by S_2 through S_k ($K \geq 1$).

Φ-Type: **[STRUCTURES × {STRUCTURES}*] → PAIRS** *Properties:* Cons.

Examples:

```
(XCONS '+ '2 '3)    ⇒  '(+ 2 3)
(XCONS 'ACONS)      ⇒  '(ACONS)
(XCONS 1 2 3)       ⇒  {ERROR: Structure expected.}
```

4.c.2. RAILS and SEQUENCES

(RCONS S_1 ... S_k**)**

Designates an otherwise completely inaccessible rail of length k whose elements are the internal structures designated by S_1 through S_k ($k \geq 0$).

Φ-Type: **[{STRUCTURES}*] → RAILS** *Properties:* Primitive; kernel; cons.

Examples:

```
(RCONS '1 '2 '3)            ⇒  '[1 2 3]
(RCONS 'A (PCONS 'B 'C))    ⇒  '[A (B . C)]
(RCONS)                     ⇒  '[]
(= (RCONS) (RCONS))         ⇒  $F
(= ↓(RCONS) ↓(RCONS))       ⇒  $T
(RCONS 1 2 3)               ⇒  {ERROR: Structure expected.}
```

(SCONS E_1 ... E_k)

Designates the sequence of length k of objects (internal or external) designated by E_1 through E_k ($k \geq 0$); returns an otherwise completely inaccessible normal-form designator (rail) of that sequence. Note that sequence identity is as in mathematics: two sequences are the same if, and only if, they consist of the same elements in the same order.

Φ-Type: *[{OBJECTS}*] → SEQUENCES* *Properties:* Primitive; kernel; cons.

Examples:

```
(SCONS 1 2 3)                        ⇒  [1 2 3]
(SCONS '1 '2 '3)                     ⇒  ['1 '2 '3]
(SCONS 'A (+ 2 2))                   ⇒  ['A 4]
['A (+ 2 2)]                         ⇒  ['A 4]
(SCONS)                              ⇒  []
(= (SCONS) (SCONS))                  ⇒  $T
(= ↑(SCONS) ↑(SCONS))                ⇒  $F
(LET [[X [1 2]]] (= X (SCONS . X)))  ⇒  $T
(LET [[X [1 2]]] (= ↑X ↑(SCONS . X))) ⇒ $F
```

(PREP *E* *VEC*)

Designates a vector (of the same type as designated by *VEC*) whose first element is the object designated by *E*, and whose first tail is the vector designated by *VEC*. When *VEC* designates a sequence, (PREP *E* *VEC*) returns an otherwise inaccessible rail whose first tail is the same rail as that to which *VEC* normalizes (i.e., it returns an inaccessible but not completely inaccessible rail). When *VEC* designates a rail, (PREP *E* *VEC*) returns the handle of an otherwise inaccessible rail whose first tail is the rail which *VEC* designates. Note that 'PREP' — short for 'prepend' — is pronounced in a way that connotes alligators.

Φ-Types: *[OBJECTS × SEQUENCES] → SEQUENCES* *Properties:* Primitive; kernel; cons.
[STRUCTURES × RAILS] → RAILS

Examples:

```
(PREP 10 [20 30])     ⇒  [10 20 30]
(PREP 'A '[B C])      ⇒  '[A B C]
(PREP #S "pain")      ⇒  "Spain"
(PREP [$T] [$F])      ⇒  [[$T] $F]
(PREP 10 '[20 30])    ⇒  {ERROR: Structure expected.}
(PREP '10 [20 30])    ⇒  ['10 20 30]
(PREP 1 2)            ⇒  {ERROR: Vector expected.}
```

(LENGTH *VEC*)

Designates the number of elements in the rail or sequence designated by *VEC*.

Φ-Type: *[{RAILS ∪ SEQUENCES}] → NUMBERS* *Properties:* Primitive.

Examples:

```
(LENGTH '[A B C])     ⇒  3
(LENGTH (SCONS))      ⇒  0
(LENGTH "Five")       ⇒  4
(LENGTH 3)            ⇒  {ERROR: Vector expected.}
```

(NTH *N* *VEC*)

When *N* designates the number k, (NTH *N* *VEC*) designates the k'th element of the rail or sequence designated by *VEC*. Vector elements are numbered starting at 1, not 0; therefore k may range from 1 to the length of the designation of *VEC*.

Φ-Types: *[NUMBERS × RAILS] → STRUCTURES* *Properties:* Primitive; kernel.
[NUMBERS × SEQUENCES] → OBJECTS

Examples:

```
(NTH 1 [(+ 5 5) 20 30])   ⇒  10
(NTH 2 ['10 '20 '30])     ⇒  '20
(NTH 3 '[10 20 30])       ⇒  '30
(NTH 4 "Eight")           ⇒  #h
(NTH 2 [10])              ⇒  {ERROR: Index too large.}
(NTH '2 [10 20 30])       ⇒  {ERROR: Number expected.}
(NTH 1 10)                ⇒  {ERROR: Vector expected.}
```

(TAIL *N VEC*)

Designates the *N*'th tail of the rail or sequence designated by *VEC* (where *N* may range from 0 to the length of *VEC*). In general, the *k*'th tail of a vector of length K is that vector consisting of the (*k*+1)'th through K'th element; thus the 0'th tail of *A* is identically *A*. If (TAIL *N VEC*) designates a sequence, it will return the *N*'th tail of the rail to which *VEC* normalizes.

Φ-Types: *[NUMBERS × RAILS] → RAILS* *Properties:* Primitive; kernel.
[NUMBERS × SEQUENCES] → SEQUENCES

Examples:

```
(TAIL 2 [10 20 30 40])                ⇒   [30 40]
(TAIL 1 (CDR '(RCONS 'A 'B 'C)))      ⇒   '['B 'C]
(LET [[X '[A B]]] (= X (TAIL 0 X)))   ⇒   $T
(LETSEQ [[X [2 3]]
         [Y (PREP 1 X)]]
   (= ↑X ↑(TAIL 1 Y)))                ⇒   $T
(TAIL 1 [1])                          ⇒   []
(TAIL 4 "Kangaroo")                   ⇒   "aroo"
(TAIL 3 [1 2])                        ⇒   {ERROR: Index too large.}
(TAIL $F [1 2])                       ⇒   {ERROR: Number expected.}
(TAIL 1 #C)                           ⇒   {ERROR: Vector expected.}
```

(EMPTY *VEC*)

True just in case *VEC* designates an empty rail or sequence; false in case *VEC* designates a non-empty rail or sequence; error otherwise. Note that (EMPTY *VEC*) will return $F even if *VEC* designates an infinite vector (in contrast with LENGTH).

Φ-Type: *[{RAILS ∪ SEQUENCES}] → TRUTH-VALUES* *Properties:* Primitive; kernel.

Examples:

```
(EMPTY [])                        ⇒   $T
(EMPTY '[])                       ⇒   $T
(EMPTY '[A B C])                  ⇒   $F
(EMPTY (SCONS))                   ⇒   $T
(EMPTY (RCONS))                   ⇒   $T
(EMPTY "No")                      ⇒   $F
(LET [[X (RCONS '1)]]
   (BLOCK (REPLACE (TAIL 1 X) X)
          (EMPTY X)))             ⇒   $F
(EMPTY '(A . B))                  ⇒   {ERROR: Vector expected.}
```

(UNIT *VEC*)
(DOUBLE *VEC*)

True just in case the vector designated by *VEC* is of length 1 or 2, respectively. Note that each of these forms will return $F even if *VEC* designates an infinite vector (i.e., they are defined in terms of EMPTY, not LENGTH).

Φ-Type: *[{RAILS ∪ SEQUENCES}] → TRUTH-VALUES* *Properties:* Kernel.

Examples:

```
(UNIT '[A])                   ⇒   $T
(DOUBLE (REST [10 20 30]))    ⇒   $T
(DOUBLE "Two")                ⇒   $F
(UNIT 1)                      ⇒   {ERROR: Vector expected.}
```

(FOOT *VEC*)

Designates the empty vector that is the last tail of the vector designated by *VEC*. If *VEC* designates a sequence, (FOOT VEC) will return the last tail of the rail to which *VEC* normalizes. FOOT is primarily useful in the (destructive) extending of vectors (see the definition of CONCATENATE, for example).

Φ-Types: *[RAILS] → RAILS*
[SEQUENCES] → SEQUENCES

Examples:

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

```
(LET [[X (SCONS 10 20)]]
   (BLOCK (REPLACE (FOOT ↑X) '[30 40])
          X))                              ⇒   [10 20 30 40]
```

(REST *VEC*)

Designates the first tail of the vector designated by *VEC*. REST plays the role in 3-LISP that CDR plays in standard LISPs when used on lists signifying enumerations.

Φ-*Types:* *[RAILS] → RAILS* *Properties:* Kernel.
[SEQUENCES] → SEQUENCES

Examples:

```
(REST [1 2 3])     ⇒   [2 3]
(REST 1)           ⇒   {ERROR: Vector expected.}
```

(1ST *VEC*)
(2ND *VEC*)
(3RD *VEC*)
(4TH *VEC*)
(5TH *VEC*)
(6TH *VEC*)

These forms designate, respectively, the first, second, third, fourth, fifth, and sixth elements of the vector designated by *VEC*. In case *VEC* designates a sequence, each returns the Kth element of the rail to which *VEC* normalizes ($1 \le K \le 6$). Defined to be (NTH 1 *VEC*), (NTH 2 *VEC*), etc.

Φ-*Type:* *[SEQUENCES] → OBJECTS* *Properties:* Kernel (1ST and 2ND only).
[RAILS] → STRUCTURES

Examples:

```
(3RD [10 20 30 40])      ⇒   30
(1ST (PREP 'A '[B C]))   ⇒   'A
(2ND [1])                ⇒   {ERROR: Index too large.}
```

(MEMBER *E VEC*)

True when the object designated by *E* is an element of the vector designated by *VEC*. If (MEMBER *E VEC*) is true, it is guaranteed to return; if not, it will terminate only if the vector designated by *VEC* is finite. Note: Since MEMBER is defined in terms of =, it can't be used over sequences of functions.

Φ-*Type:* *[OBJECTS × SEQUENCES] → TRUTH-VALUES* *Properties:* Kernel.
[STRUCTURES × RAILS] → TRUTH-VALUES

Examples:

```
(MEMBER 1 [2 3 4])             ⇒   $F
(MEMBER 3 [1 1 2 (+ 1 2)])     ⇒   $T
(MEMBER '2 '[1 2 3])           ⇒   $T
(MEMBER 2 ['1 '2 '3])          ⇒   $F
(MEMBER '[] '[[A] [] [B]])     ⇒   $F
(MEMBER [] [[1] [] [2]])       ⇒   $T
(MEMBER 1 2)                   ⇒   {ERROR: Vector expected.}
(MEMBER * [+ - * /])           ⇒   {ERROR: = not defined over functions.}
```

(VECTOR-CONSTRUCTOR *TEMPLATE*)

Designates the RCONS or SCONS procedure, depending on whether *TEMPLATE* designates an internal structure or external object, respectively. VECTOR-CONSTRUCTOR is primarily useful in the terminating clause of a recursive definition defined over general vectors (see the definition of MAP, for example).

Φ-*Type:* *[OBJECTS] → FUNCTIONS* *Properties:* Kernel.

Examples:

```
(VECTOR-CONSTRUCTOR '[])       ⇒   {Simple RCONS closure}
((VECTOR-CONSTRUCTOR '[]))     ⇒   '[]
(VECTOR-CONSTRUCTOR 100)       ⇒   {Simple SCONS closure}
((VECTOR-CONSTRUCTOR 100))     ⇒   []
(VECTOR-CONSTRUCTOR ↑↑1)       ⇒   '[]
```

(MAP *FUN* V_1 V_2 ... V_k)

Designates the vector obtained by applying the function designated by *FUN* (of arity *k*) to successive elements of the vectors designated by V_1 through V_k. The vectors V_1 through V_k should be of equal length.

Φ-*Type:* *[FUNCTIONS × {SEQUENCES}*] → SEQUENCES* *Properties:* Kernel; cons.
[FUNCTIONS × {RAILS}] → RAILS*

Examples:
```
(MAP 1+ [2 3 4])                 ⇒  [3 4 5]
(MAP * [1 2 3] [1 2 3])          ⇒  [1 4 9]
(MAP EF [$T $F] [1 2] [3 4])     ⇒  [1 4]
(MAP CAR [])                     ⇒  []
(MAP UP '[1 A $T])               ⇒  '['1 'A '$T]
(MAP 1+ [1 2 3] [4 5 6])         ⇒  {ERROR: Too many arguments.}
(MAP 1 [1 2 3])                  ⇒  {ERROR: Not a function.}
(MAP 1+ 100)                     ⇒  {ERROR: Vector expected.}
```

(COPY-VECTOR *VEC*)

If *VEC* designates a rail, (COPY-VECTOR *VEC*) designates an otherwise completely inaccessible rail whose elements are the elements of the rail designated by *VEC*. If *VEC* designates a sequence, (COPY-VECTOR *VEC*) designates the same sequence as *VEC*, but returns an otherwise completely inaccessible designator (rail) of it. Note that when *VEC* designates a sequence, (SCONS . *VEC*) could be used to achieve the same effect.

Φ-*Types:* *[RAILS] → RAILS* *Properties:* Cons.
[SEQUENCES] → SEQUENCES

Examples:
```
(COPY-VECTOR '[A B C])               ⇒  '[A B C]
(COPY-VECTOR [])                     ⇒  []
(LET [[Y [1 2 3]]]
   [(= Y (COPY-VECTOR Y))
    (= ↑Y ↑(COPY-VECTOR Y))])        ⇒  [$T $F]
```

(CONCATENATE R_1 R_2)

CONCATENATE replaces the foot of the rail designated by R_1 with the rail designated by R_2. More formally, if L_1 and L_2 are the lengths of the rails designated by R_1 and R_2, respectively, then (CONCATENATE R_1 R_2) designates a rail of length L_1+L_2, whose first L_1 elements are the elements of the rail designated by R_1, whose L_2'th tail is the rail R_2. The rail to which R_1 normalizes is affected, so CONCATENATE should be used with extreme caution; normally **APPEND** will do the job.

Φ-*Types:* *[RAILS × RAILS] → RAILS* *Properties:* Smash.

Examples:
```
(CONCATENATE '[A] '[B C])                ⇒  '[A B C]
(LET [[X (RCONS)]]
   (BLOCK (CONCATENATE X '[NEW TAIL])
          X))                            ⇒  '[NEW TAIL]
(LET [[X '[1 2 3]]
      [Y '[4 5]]]
   (BLOCK (CONCATENATE X Y)
          [^ Y]))                        ⇒  ['[1 2 3 4 5]
                                             '[4 5]]
```

(APPEND V_1 V_2)

If L_1 and L_2 are the respective lengths of the vectors designated by V_1 and V_2, (APPEND V_1 V_2) designates the vector of length L_1+L_2 whose first L_1 elements are the elements of V_1, and whose remaining L_2 elements are those of V_2. Both vectors must be of the same type. The vector to which V_2 normalizes is not copied (i.e., V_2 is accessible from the result).

Φ-*Types:* *[RAILS × RAILS] → RAILS* *Properties:* Cons.
[SEQUENCES × SEQUENCES] → SEQUENCES

Examples:
```
(APPEND [1 2 3] [4 5 6])           ⇒  [1 2 3 4 5 6]
(APPEND '[] '[A B C])              ⇒  '[A B C]
(LET [[X '[M N]]] (APPEND X X))    ⇒  '[M N M N]
```

```
(LETSEQ [[X '[M N]] [Y (APPEND X X)]]
     (= X (TAIL 2 Y)))                        ⇒     $T
(LET [[X [1 2]] [Y [3 4]]]
   (BLOCK (APPEND X Y)
          X))                                 ⇒     [1 2]
(APPEND "Of shoes" " and ships")              ⇒     "Of shoes and ships"
(APPEND 1 [2 3])                              ⇒     {ERROR: Vector expected.}
```

(APPEND* V_1 V_2 ... V_k)

APPEND* is a variant of APPEND that accepts multiple argument vectors. More formally, if L_i is the length of the vector designated by each V_i, (APPEND V_1 V_2 ... V_k) designates the vector of length $L_1+L_2+ ... + L_k$ whose first L_1 elements are the elements of the vector designated by V_1, and whose next L_2 elements are the elements of the vector designated by V_2, etc. ($K \geq 1$). All vectors must be of the same type. The vectors to which V_K normalizes is not copied (i.e., V_K is accessible from the result).

Φ-Types: [RAILS × {RAILS}*] → RAILS *Properties:* Cons.
[SEQUENCES × {SEQUENCES}*] → SEQUENCES

Examples:
```
(APPEND* [1 2 3] [4 5 6] [7 8 9])     ⇒ [1 2 3 4 5 6 7 8 9]
(APPEND* '[A B C])                    ⇒ '[A B C]
(LET [[X '[G O]]] (APPEND* X X X))    ⇒ '[G O G O G O]
(APPEND* "Mac" "H" "i" "n" "e")       ⇒ "MacHine"
```

(REVERSE *VEC*)

Designates a vector (of the type of the vector designated by VEC) whose elements are the same as the elements of the vector designated by VEC, except in reverse order. The resulting vector is otherwise completely inaccessible.

Φ-Types: [RAILS] → RAILS *Properties:* Cons.
[SEQUENCES] → SEQUENCES

Examples:
```
(REVERSE [])                          ⇒ []
(REVERSE [1 2 3])                     ⇒ [3 2 1]
(REVERSE '[[A B] [C D]])              ⇒ '[[C D] [A B]]
(LET [[X [10]]] (= X (REVERSE X)))    ⇒ $T
(LET [[X [10]]]
   (= ↑X ↑(REVERSE X)))               ⇒ $F
(LET [[Y '[A]]] (= Y (REVERSE Y)))    ⇒ $F
```

(INDEX *ELEMENT VECTOR*)

Searches the vector designated by VECTOR for an element equal to the object designated by ELEMENT, and yields the number indicating the first position in which it was found. Designates 0 if the object is not a member of the vector.

Φ-Type: [OBJECTS × VECTORS] → NUMBERS

Examples:
```
(INDEX 3 [2 3 6 1])          ⇒ 2
(INDEX 'B ['A 'B 'C])        ⇒ 2
(INDEX [10] [1 $T [10]])     ⇒ 3
(INDEX #l "Hello")           ⇒ 3
(INDEX '+ [])                ⇒ 0
```

(PUSH *ELEMENT STACK*)

Pushes the object designated by ELEMENT onto the sequence designated by STACK. Structural field side effects are involved. Returns 'OK.

Φ-Type: [OBJECTS × SEQUENCES] → ATOMS *Properties:* Smash; cons.

Examples:
```
1> (SET S [])
1= 'OK
1> (PUSH 1 S)
1= 'OK
1> (PUSH 2 S)
1= 'OK
1> S
1= [2 1]
```

(POP *STACK*)

Pops the most recently PUSHed object off of the sequence designated by *STACK*. Structural field side effects are involved. Designates the object popped off.

Φ-Type: *[SEQUENCES] → OBJECTS* *Properties:* Smash.

Examples:
```
1> (BLOCK (SET S []) (PUSH 1 S) (PUSH 2 S))
1= 'OK
1> S
1= [2 1]
1> (POP S)
1= 2
1> S
1= [1
```

4.c.3. CLOSURES

(CCONS *KIND DEF-ENV PATTERN BODY*)

Designates an otherwise inaccessible closure of the type designated by *KIND* (typically either SIMPLE or REFLECT) containing designators of the environment designated by *DEF-ENV*, the pattern designated by *PATTERN*, and the body designated by *B*. (Note that (LAMBDA MACRO ...) and (LAMBDA REFLECT! ...) both construct REFLECT-type closures.)

Properties: Primitive; kernel; cons.

Φ-Type: *[ATOMS × RAILS × STRUCTURES × STRUCTURES] → CLOSURES*

Properties: Primitive; kernel; cons.

Examples:
```
(CCONS 'X '[] 'Y 'Z)                  ⇒  '{closure: X [] Y Z}
(CCONS 'SIMPLE ↑GLOBAL '[X] 'X)       ⇒  '{closure: simple {global} [X] X}
(↓(CCONS 'SIMPLE
         ↑GLOBAL
         '[X]
         '(+ X 1)) 10)                ⇒  11
```

(PROCEDURE-TYPE *CLOSURE*)

Designates the atom that is the PROCEDURE-TYPE of the closure designated by *CLOSURE*.

Φ-Type: *[CLOSURES] → ATOMS* *Properties:* Primitive; kernel.

Examples:
```
(PROCEDURE-TYPE (CCONS 'X '[] 'Y 'Z))  ⇒  'X
(PROCEDURE-TYPE ↑+)                    ⇒  'SIMPLE
(PROCEDURE-TYPE ↑IF)                   ⇒  'REFLECT
(PROCEDURE-TYPE IF)                    ⇒  {ERROR: Closure expected.}
```

(ENVIRONMENT-DESIGNATOR *CLOSURE*)

Designates the rail that is the ENV of the closure designated by *CLOSURE*. Note that while ENVIRONMENT-DESIGNATOR is semantically flat, closures are a little confused (they contain environment *designators* instead of environments). ENVIRONMENT is almost always more appropriate.

Φ-Type: *[CLOSURES] → RAILS* *Properties:* Primitive; kernel.

Examples:
```
(ENVIRONMENT-DESIGNATOR (CCONS 'X '[] 'Y 'Z))  ⇒  '[]
(ENVIRONMENT-DESIGNATOR +)    ⇒   {ERROR: Closure expected.}
```

(ENVIRONMENT *CLOSURE*)

Designates the environment in the closure designated by *CLOSURE*.

Φ-Type: *[CLOSURES] → SEQUENCES* *Properties:* Kernel.

Examples:
```
(ENVIRONMENT (CCONS 'X '[] 'Y 'Z))  ⇒  []
(ENVIRONMENT +)                     ⇒  {ERROR: Closure expected.}
```

(PATTERN *CLOSURE*)

Designates the internal structure that is the PATTERN of the closure designated by *CLOSURE*.

Φ-*Type:* *[CLOSURES] → STRUCTURES* *Properties:* Primitive; kernel.

Examples:
```
(PATTERN (CCONS 'X '[] 'Y 'Z))      ⇒  'Y
(PATTERN ↑(LAMBDA SIMPLE [A B]
             (PCONS B A)))          ⇒  '[A B]
(PATTERN ↑NORMALISE)                ⇒  '[EXP ENV CONT]
(PATTERN +)                         ⇒  {ERROR: Closure expected.}
```

(BODY *CLOSURE*)

Designates the internal structure that is the BODY of the closure designated by *CLOSURE*.

Properties: Primitive; kernel.

Φ-*Type:* *[CLOSURES] → STRUCTURES* *Properties:* Primitive; kernel.

Examples:
```
(BODY (CCONS 'X '[] 'Y 'Z))         ⇒  'Z
(BODY ↑(LAMBDA SIMPLE [A B]
          (PCONS B A)))             ⇒  '(PCONS B A)
(PATTERN ↑REST)                     ⇒  '(TAIL 1 VECTOR)
(BODY +)                            ⇒  {ERROR: Closure expected.}
```

(REFLECTIVE *CLOSURE*)

True just in case the PROCEDURE-TYPE of the closure designated by *CLOSURE* is the atom REFLECT.

Φ-*Type:* *[CLOSURES] → TRUTH-VALUES* *Properties:* Kernel; cons.

Examples:
```
(REFLECTIVE ↑+)                     ⇒  $F
(REFLECTIVE ↑IF)                    ⇒  $T
(REFLECTIVE ↑LET)                   ⇒  $T
(REFLECTIVE (CCONS 'X '[] 'Y 'Z))   ⇒  $F
```

(DE-REFLECT *CLOSURE*)

Designates an otherwise inaccessible closure whose PROCEDURE-TYPE is the atom SIMPLE and whose other components are the same as those of the closure designated by *CLOSURE*.

Φ-*Type:* *[CLOSURES] → CLOSURES* *Properties:* Kernel.

Examples:
```
(DE-REFLECT (CCONS 'X '[] 'Y 'Z))   ⇒  '{closure: simple [] Y Z}
(DE-REFLECT ↑IF)                    ⇒  '{simple IF closure}
```

(REFLECTIFY *FUN*)

Designates a function; returns an otherwise inaccessible closure whose PROCEDURE-TYPE is the atom REFLECT and whose other components are the same as those of the closure to which *FUN* normalizes. For example, BLOCK is defined in section 8 to be (REFLECTIFY BLOCK-HELPER).

Φ-*Type:* *[FUNCTIONS] → FUNCTIONS* *Properties:* Cons.

Examples:
```
(REFLECTIFY ↓(CCONS 'X '[] 'Y 'Z))  ⇒  {closure: reflect [] Y Z}
(REFLECTIFY NORMALIZE)              ⇒  {reflective NORMALIZE closure}
```

4.c.4. ATOMS

(ACONS)

Designates a nameless and otherwise inaccessible atom.

Φ-*Type:* *[ ] → ATOMS* *Properties:* Primitive; cons.

Examples:
```
(ACONS)                  ⇒  {atom}
(= (ACONS) (ACONS))      ⇒  $F
```

4.c.5. TYPING

(TYPE *A*)

Designates the atom associated with the type of the object designated by *A* (chosen from the standard 15).

Φ-*Type:* ***[OBJECTS] → ATOMS*** *Properties:* Primitive; kernel.

Examples:
```
(TYPE 3)                      ⇒ 'NUMBER
(TYPE '3)                     ⇒ 'NUMERAL
(TYPE $T)                     ⇒ 'TRUTH-VALUE
(TYPE '$F)                    ⇒ 'BOOLEAN
(TYPE #A)                     ⇒ 'CHARACTER
(TYPE '#4)                    ⇒ 'CHARAT
(TYPE [1 2 3])                ⇒ 'SEQUENCE
(TYPE '[1 2 3])               ⇒ 'RAIL
(TYPE +)                      ⇒ 'FUNCTION
(TYPE ↑+)                     ⇒ 'CLOSURE
(TYPE PRIMARY-STREAM)         ⇒ 'STREAM
(TYPE ↑PRIMARY-STREAM)        ⇒ 'STREAMER
(TYPE '(= 2 3))               ⇒ 'PAIR
(TYPE 'A)                     ⇒ 'ATOM
(TYPE ''3)                    ⇒ 'HANDLE
(TYPE '''''''?)               ⇒ 'HANDLE
```

(ATOM *E*)
(BOOLEAN *E*)
(CHARACTER *E*)
(CHARAT *E*)
(CLOSURE *E*)
(FUNCTION *E*)
(HANDLE *E*)
(NUMBER *E*)
(NUMERAL *E*)
(PAIR *E*)
(RAIL *E*)
(SEQUENCE *E*)
(STREAM *E*)
(STREAMER *E*)
(TRUTH-VALUE *E*)

Each of the fifteen type predicates are true of elements of each of fifteen semantic categories, and false of all others. Specifically, (ATOM *E*) is true iff *E* designates an atom (and similarly for the others).

Φ-*Type:* ***[OBJECTS] → TRUTH-VALUES*** *Properties:* Kernel (ATOM, PAIR, RAIL, HANDLE only).

Examples:
```
(ATOM 'A)                ⇒ $T
(PAIR '(1ST '[A B]))     ⇒ $T
(FUNCTION +)             ⇒ $T
(CLOSURE '+)             ⇒ $F
```

(VECTOR *E*)

True if, and only if, *E* designates either a rail or a sequence; false otherwise.

Φ-*Type:* ***[OBJECTS] → TRUTH-VALUES***

Examples:
```
(VECTOR [1 2 3])         ⇒ $T
(VECTOR '[A B])          ⇒ $T
(VECTOR '(1 2 3))        ⇒ $F
(VECTOR "String")        ⇒ $T
```

(INTERNAL *E*)
(EXTERNAL *E*)

(INTERNAL *E*) is true if, and only if, *E* designates an *internal* structure such as a numeral or a rail; false otherwise. Similarly, (EXTERNAL *E*) is true just in case *E* designates an *external* structure (i.e., an *abstraction*) such as a number or a sequence; false otherwise.

Φ-Type: *[OBJECTS] → TRUTH-VALUES* *Properties:* Kernel (EXTERNAL only).

Examples:
```
(EXTERNAL 123)         ⇒  $T
(INTERNAL (+ 2 2))     ⇒  $F
(EXTERNAL +)           ⇒  $T
(INTERNAL '+)          ⇒  $T
(INTERNAL ↑+)          ⇒  $T
```

(CHARACTER-STRING *E*)

True if, and only if, E designates a sequence of one or more characters; false otherwise.

Φ-Type: *[OBJECTS] → TRUTH-VALUES*

Examples:
```
(CHARACTER-STRING "Hello")           ⇒ $T
(CHARACTER-STRING [#A #B #C])        ⇒ $T
(CHARACTER-STRING #X)                ⇒ $F
(CHARACTER-STRING "")                ⇒ $F
(CHARACTER-STRING '[1 2 3])          ⇒ $F
(CHARACTER-STRING (PREP 1 "2"))      ⇒ $F
```

4.c.6. IDENTITY

(= E_1 E_2 ... E_K)

When K is 2, true if E_1 and E_2 designate the same object; false otherwise. However, an error will be detected if *both* E_1 and E_2 designate functions. When both E_1 and E_2 designate sequences, corresponding elements are compared (using =) from left to right until it can be established that the two sequences differ, or until an error is detected. Consequently, (= E_1 E_2) may fail to terminate when E_1 and E_2 designate infinite sequences (or sequences containing infinite sequences). Note that although equality is defined over closures, it is too fine-grained to be used for function identity. When K is greater than 2, E_I will not be compared to E_1 unless E_1 through E_{I-1} have been determined to all designate the same object.

Φ-Type: *[OBJECTS × OBJECTS × {OBJECTS}*] → TRUTH-VALUES* *Properties:* Primitive; kernel.

Examples:
```
(= 3 (+ 1 2))              ⇒  $T
(= 5 '5)                   ⇒  $F
(= '5 '5)                  ⇒  $T
(= $F $F $F $F)            ⇒  $T
(= [10 20] [10 20])        ⇒  $T
(= '[10 20] '[10 20])      ⇒  $F
(= ['10 '20] '[10 20])     ⇒  $F
(= '[10 20] ['10 '20])     ⇒  $F
(= CAR CDR)                ⇒  {ERROR: = not defined over functions.}
(= CAR 3)                  ⇒  $F
(= [+ 2] [+ 3])            ⇒  {ERROR: = not defined over functions.}
(= [2 +] [3 +])            ⇒  $F
(= + 1 +)                  ⇒  $F
(= + + 1)                  ⇒  {ERROR: = not defined over functions.}
```

(ISOMORPHIC E_1 E_2)

True if E_1 and E_2 designate similar structures; false otherwise. When either E_1 or E_2 designates an external structure ISOMORPHIC behaves just like =. Otherwise, two internal structures are isomorphic if they are = or have isomorphic corresponding components. ISOMORPHIC may fail to terminate on circular structures.

Φ-Type: `[ OBJECTS × OBJECTS ] → TRUTH-VALUES`

Examples:

```
(ISOMORPHIC '5 '5)                         ⇒  $T
(ISOMORPHIC '[10 20] '[10 20])             ⇒  $T
(ISOMORPHIC '[10 20] ['10 '20])            ⇒  $F
(ISOMORPHIC ↑CAR ↑CDR)                     ⇒  $F
(ISOMORPHIC '(A . B) '(A . B))             ⇒  $T
(ISOMORPHIC ''[X] ''[X])                   ⇒  $T
(ISOMORPHIC ↑(LAMBDA SIMPLE [X] X)
            ↑(LAMBDA SIMPLE [X] X))        ⇒  $T
```

4.c.7. ARITHMETIC OPERATIONS

(+ N_1 N_2 ... N_k)
(* N_1 N_2 ... N_k)

Designate, respectively, the sum and product of the numbers designated by N_1 through N_k. `(+)` designates 0, and `(*)` designates 1.

Φ-Type: `[ {NUMBERS}* ] → NUMBERS` *Properties:* Primitive.

Examples:

```
(* 2 2 2 2)      ⇒ 16
(+ 1 3 5)        ⇒ 9
(+ 3)            ⇒ 3
(* 3)            ⇒ 3
(+)              ⇒ 0
(*)              ⇒ 1
(+ '1 '2)        ⇒ {ERROR: Number expected.}
```

(- N_1 N_2 ... N_k)

Designates the difference of the numbers designated by N_1 through N_k. k must be at least 1. Specifically, `(- N)` is equivalent to `(- 0 N)`, and `(- N1 N2 ... Nk)` is equivalent to `(- N1 (+ N2 ... Nk))`.

Φ-Type: `[ NUMBER × {NUMBERS}* ] → NUMBERS` *Properties:* Primitive.

Examples:

```
(- 100 2)          ⇒ 98
(- 3)              ⇒ -3
(- 10 20)          ⇒ -10
(- 9 1 3 5)        ⇒ 0
(- 9 (+ 1 3 5))    ⇒ 0
(-3)               ⇒ {ERROR: Not a function.}
(- 0 $T)           ⇒ {ERROR: Number expected.}
```

(/ N_1 N_2)

Designates the quotient of the numbers designated by N_1 and N_2. `(/ N1 N2)` will cause an error if N_2 designates zero. Currently, arithmetic is defined only on integers; ultimately we intend to define full rational (or repeating fraction) arithmetic, with no upper limit on numeral size, and no limit on precision.

Φ-Type: `[ NUMBERS × NUMBERS ] → NUMBERS` *Properties:* Primitive.

Examples:

```
(/ 10 3)         ⇒ 3
(/ -10 3)        ⇒ -3
(/ 10 -3)        ⇒ -3
(/ -10 -3)       ⇒ 3
(/ 100 0)        ⇒ {ERROR: Division by zero.}
```

(REMAINDER N_1 N_2)

Designates the remainder upon dividing N_1 by N_2; error if N_2 designates zero. The sign of a non-zero remainder is that of the first argument.

Φ-Type: `[ NUMBER × NUMBERS ] → NUMBERS`

Examples:

```
(REMAINDER 10 3)      ⇒   1
(REMAINDER 10 -3)     ⇒   1
(REMAINDER -10 -3)    ⇒   -1
(REMAINDER -10 -3)    ⇒   -1
(REMAINDER 10 0)      ⇒   {ERROR: Division by zero.}
```

(1+ *N*)
(1- *N*)

Designates the number one greater or one less than the number designated by *N*, respectively.

Φ-Type: *[NUMBERS] → NUMBERS*

Examples:
```
(1+ 20)          ⇒   21
(MAP 1- [2 3 4]) ⇒   [1 2 3]
```

(< N_1 N_2 ... N_k)
(<= N_1 N_2 ... N_k)
(> N_1 N_2 ... N_k)
(>= N_1 N_2 ... N_k)

True if, and only if, the number designated by N_1 is less than the number designated by N_2, the number designated by N_2 is less than the number designated by N_3, etc. Similarly for the others, except that the relationship is that of being less than or equal (<=), greater than (>), or greater than or equal (>=). In all cases, *k* must be at least 2.

Φ-Type: *[NUMBERS × NUMBERS × {NUMBERS}*] → TRUTH-VALUES* *Properties:* Primitive.

Examples:
```
(< 2 3)              ⇒  $T
(>= 5 4 4 2 -1 -7)   ⇒  $T
(<= 99 1 '1)         ⇒  {ERROR: Number expected.}
(> 100 1000)         ⇒  $F
```

(ABS *N*)

Designates the absolute value of the number designated by *N*.

Φ-Type: *[NUMBER] → NUMBERS*

Examples:
```
(ABS 100)     ⇒    100
(ABS -100)    ⇒    100
(ABS 0)       ⇒    0
(ABS '1)      ⇒    {ERROR: Number expected.}
```

(MIN N_1 N_2 ... N_k)
(MAX N_1 N_2 ... N_k)

Designate, respectively, the minimum and maximum of the numbers designated by N_1 through N_k ($K \geq 1$).

Φ-Type: *[NUMBERS × {NUMBERS}*] → NUMBERS*

Examples:
```
(MIN 3 1 4)    ⇒   1
(MIN 0 1 -7)   ⇒   -7
(MAX 4)        ⇒   4
```

(ODD *N*)
(EVEN *N*)

True if *N* designates an odd or even number, respectively.

Φ-Type: *[NUMBERS] → TRUTH-VALUES*

Examples:
```
(ODD 100)    ⇒   $F
(EVEN 100)   ⇒   $T
(ODD -1)     ⇒   $T
```

(ZERO *N*)
(NEGATIVE *N*)
(POSITIVE *N*)
(NON-NEGATIVE *N*)

True if the number *N* designates is equal to, less than, greater than, or greater than or equal to zero, respectively.

Φ-Type: ***[NUMBERS] → TRUTH-VALUES***

Examples:
```
(ZERO 1)          ⇒  $F
(NEGATIVE -1)     ⇒  $T
(POSITIVE 0)      ⇒  $F
(NON-NEGATIVE 0)  ⇒  $T
```

(N_1 N_2)**

Designates the N_2-fold product of the number designated by N_1 with itself. N_2 must designate a non-negative number.

Φ-Type: ***[NUMBERS × NUMBERS] → NUMBERS***

Examples:
```
(** 2 10)   ⇒  1024
(** 10 0)   ⇒  1
(** -5 3)   ⇒  -125
```

4.c.8. PROCEDURE DEFINITION and VARIABLE BINDING

(DEFINE *LABEL FUN*)

Establishes a binding of the atom *LABEL* (*not* the designation of that atom — i.e., *LABEL* is in an intensional context) to the function designator that results from normalizing *FUN*. Unlike SET, DEFINE normalizes *FUN* in an environment in which *LABEL* will ultimately be bound to the result of the normalization, to facilitate recursion. In other words, (DEFINE *LABEL FUN*) establishes *LABEL* as the *public* name for the function designated by *FUN*, and also enables *FUN* to use *LABEL* as its own internal name for itself. Returns a handle to *LABEL*.

Properties: Smash-env; abnormal.

Macro:

```
(DEFINE LABEL FUN)
≡> (BLOCK (SET LABEL
                (Y-OPERATOR (LAMBDA SIMPLE [LABEL] FUN)))
          'LABEL)
```

Examples:

```
1> (DEFINE SQUARE (LAMBDA SIMPLE [N] (* N N)))
1= 'SQUARE
1> (DEFINE FACTORIAL
     (LAMBDA SIMPLE [N]
        (IF (= N 0) 1 (* N (FACTORIAL (1- N))))))
1= 'FACTORIAL
1> (FACTORIAL 6)
1= 120
```

(Y-OPERATOR *FUN*)

(Y-OPERATOR *FUN*) designates a function with the property that (*FUN* (Y-OPERATOR *FUN*)) also designates that same function. In other words, (Y-OPERATOR *FUN*) is a *fixed point* of the function designated by *FUN*. *FUN* must designate a mapping from functions to functions. This *fixed point operator* is used in defining recursive procedures (see the definition of DEFINE).

Φ-Type: [*FUNCTIONS*] → *FUNCTIONS*

Examples:

```
1> (SET FACTORIAL
     (Y-OPERATOR
        (LAMBDA SIMPLE [SELF]
           (LAMBDA SIMPLE [N]
              (IF (= N 0) 1 (* N (SELF (1- N))))))))
1= 'OK
1> (FACTORIAL 6)
1= 120
```

(Y*-OPERATOR F_1 F_2 ... F_k)

Y*-OPERATOR is a generalization of Y-OPERATOR that is useful in defining multiple mutually-recursive procedures.

Φ-Type: [{*FUNCTIONS*}*] → *SEQUENCES*

Examples:

```
1> (SET EVEN&ODD
     (Y*-OPERATOR
        (LAMBDA SIMPLE [EVEN ODD]
           (LAMBDA SIMPLE [N]
              (IF (= N 0) $T (ODD (1- N)))))
        (LAMBDA SIMPLE [EVEN ODD]
           (LAMBDA SIMPLE [N]
              (IF (= N 0) $F (NOT (EVEN N)))))))
1= 'OK
1> ((1ST EVEN&ODD) 2)
1= $T
1> ((2ND EVEN&ODD) 2)
1= $F
```

(LAMBDA *TYPE PAT BODY*)

Informally, an expression of the form (LAMBDA *TYPE PAT BODY*) designates the function of type *TYPE* (typically SIMPLE, REFLECT, or MACRO) that is signified by the lambda abstraction of the formal parameters in pattern *PAT* over the expression *BODY*. LAMBDA is intensional in its second and third argument positions: neither *PAT* nor *BODY* is normalized. More formally, (LAMBDA *TYPE PAT BODY*) designates the result of applying the function designated by *TYPE* to three arguments: a designator of the current environment, and the two expressions *PAT* and *BODY* (un-normalized).

Properties: Kernel; cons; abnormal.

Examples:

```
1> (LAMBDA SIMPLE [A B] (* A B))
1= {closure: SIMPLE {global} [A B] (* A B)}
1> ((LAMBDA SIMPLE [N] (+ N N)) 4)
1= 8
1> ((LAMBDA REFLECT [ARGS ENV CONT] ARGS) . XXX)
2= 'XXX
```

(SIMPLE *DEF-ENV PAT BODY*)

This procedure, together with REFLECT, MACRO, REFLECT!, E-MACRO, and E-REFLECT, are most useful as the *TYPE* specification in the context (LAMBDA *TYPE PAT BODY*). SIMPLE is used to define *simple procedures*. When a procedure call (*FOO* . *ARGS*) is normalized at level *N* and *FOO* designates a simple procedure, the sequence of events will be as follows: *ARGS* will be normalized in the current level *N* environment; the defining environment, *DEF-ENV*, will be extended by matching the pattern, *PAT*, against the arguments; finally, the body, *BODY*, will be normalized at level *N* in this new environment.

Φ-Type: *[RAILS × STRUCTURES × STRUCTURES] → FUNCTIONS* *Properties:* Kernel; cons.

Examples:

```
(SIMPLE '[['X '1]] '[] 'X)     ⇒ {closure: SIMPLE [['X '1]] [] X}
((SIMPLE '[['X '1]] '[] 'X))   ⇒ 1
((SIMPLE ↑GLOBAL
         '[X]
         '(+ X 2)) 99)         ⇒ 101
((SIMPLE '[] '[X] '(ACONS)))   ⇒ {ERROR: Unbound variable ACONS.}
```

(REFLECT *DEF-ENV PAT BODY*)

REFLECT is used to define *reflective procedures*. When a procedure call (*FOO* . *ARGS*) is normalized at level *N* and *FOO* designates a reflective procedure, the sequence of events will be as follows: the defining environment, *DEF-ENV*, will be extended by matching the pattern, *PAT*, against a designator of the un-normalized *ARGS*, the level *N* environment, and the current level *N+1* processor continuation; lastly, the body, *BODY*, will then be normalized at level *N+1* in this new environment.

Φ-Type: *[RAILS × STRUCTURES × STRUCTURES] → FUNCTIONS* *Properties:* Cons.

Examples:

```
1> (SET REFLECT-TEST
        (LAMBDA REFLECT [ARGS ENV CONT]
           (BLOCK (SET STASH ARGS) (CONT ''OK))))
1= 'OK
1> (REFLECT-TEST + (+ 2 2))
1= 'OK
1> STASH
1= '[+ (+ 2 2)]
1> (REFLECT-TEST . +)
1= 'OK
1> STASH
1= '+
```

(REFLECT! *DEF-ENV PAT BODY*)

REFLECT! is very similar to REFLECT except that the arguments to the reflective procedure are normalized before being matched against the pattern.

Φ-Type: *[RAILS × STRUCTURES × STRUCTURES] → FUNCTIONS* *Properties:* Cons.

Examples:

```
1> (SET REFLECT!-TEST
      (LAMBDA REFLECT! [ARGS ENV CONT]
         (BLOCK (SET STASH ARGS) (CONT ''OK))))
1= 'OK
1> (REFLECT!-TEST + (+ 2 2))
1= 'OK
1> STASH
1= '[{simple + closure} 4]
1> (REFLECT!-TEST . +)
1= 'OK
1> STASH
1= '{simple + closure}
```

(MACRO *DEF-ENV PAT BODY*)

When a procedure call (FOO . ARGS) is normalized (at level *N*) and FOO designates a macro procedure, the sequence of events will be as follows: the arguments to the procedure will not be normalized; the defining environment will be extended by matching the pattern against a designator of the un-normalized arguments; the body will be normalized in this new environment; finally, the result of this normalization will be re-normalized in the original environment.

Φ-Type: [RAILS × STRUCTURES × STRUCTURES] → FUNCTIONS *Properties:* Cons.

Examples:

```
1> (SET MACRO-TEST
      (LAMBDA MACRO ARGS
         (BLOCK (SET STASH ARGS) ARGS)))
1= 'OK
1> (MACRO-TEST + (+ 2 2))
1= [{simple + closure} 4]
1> STASH
1= '[+ (+ 2 2)]
1> (MACRO-TEST . +)
1= {simple + closure}
1> STASH
1= '+
```

(REBIND *VAR BIND ENV*)

Modifies the environment designated by ENV to contain a binding of the internal structure designated by VAR to the internal structure designated by BIND. If the structure designated by VAR is already bound, that binding will be modified in place; if not, a new binding of the structure designated by VAR to the structure designated by BIND will be added to the foot of the environment designated by ENV. Environments generated by the 3-LISP processor consist only of atoms bound to normal-form structures, so that VAR should designate an atom and BIND a normal-form internal structure if ENV is intended to continue to designate a well-formed 3-LISP environment. Returns 'OK.

Φ-Type: [STRUCTURES × STRUCTURES × SEQUENCES] → ATOMS *Properties:* Cons; smash.

Examples:

```
(LET [[ENV [['X '1] ['Y '2]]]]
   (BLOCK (REBIND 'Y ↑(+ 2 3) ENV)
          (REBIND 'Z '$T ENV)
          ENV))                    ⇒   [['X '1] ['Y '5] ['Z '$T]]
```

(SET *VAR BINDING*)

SET alters the current environment's binding of the atom VAR to be the result of normalizing BINDING (in the current environment). Note that the first argument, VAR, is not normalized. Returns 'OK.

Properties: Smash-env; abnormal.

Examples:

```
1> (SET X (+ 2 2))
1= 'OK
1> X
1= 4
1> (SET X (+ X X))
1= 'OK
```

(SETREF *VAR BINDING*)

SETREF is a variant of SET in which both *VAR* and *BINDING* are normalized. Returns 'OK.
Properties: Smash-env.

Φ-Type: *[ATOMS × OBJECTS] → ATOMS* *Properties:* Smash-env.

Examples:
```
1> (SET X 'Y)
1= 'OK
1> (SETREF X (* 2 2))
1= 'OK
1> Y
1= 4
```

(BINDING *VAR ENV*)

Designates the binding of the internal structure designated by *VAR* in the environment designated by *ENV*. The 3-LISP processor will, on its own, only establish normal-form bindings for atoms, so *VAR* should designate an atom unless the user provides his or her own environment structure (in which case *BINDING* can be used as a 3-LISP analog of LISP 1.5's ASSOC).

Φ-Type: *[STRUCTURES × SEQUENCES] → STRUCTURES* *Properties:* Kernel.

Examples:
```
(BINDING 'Y [['X '1] ['Y '2] ['Z '3]])  ⇒  '2
(BINDING 'NORMALIZE GLOBAL)             ⇒  '{simple NORMALIZE closure}
(LET [[X (+ 1 2)]]
   ((LAMBDA REFLECT [ARGS ENV CONT]
        (CONT (BINDING 'X ENV)))))      ⇒  3
```

(BIND *PATTERN ARGS ENV*)

Designates an environment obtained by augmenting the environment designated by *ENV* with the variable bindings that result from the matching of the pattern structure designated by *PATTERN* against the argument structure designated by *ARGS*. A pattern consisting of a single atom will match any argument structure directly; this results in the atom becoming bound to the entire argument structure. This basic matching process is extended to rail patterns in the usual way: pattern and argument rails must match on an element by element basis. The designator of the old environment is always a tail of the result.

Φ-Type: *[STRUCTURES × STRUCTURES × SEQUENCES] → SEQUENCES* *Properties:* Kernel; cons.

Examples:
```
(BIND 'X '2 [['Y '1]])                  ⇒  [['X '2] ['Y '1]]
(BIND '[X] '[2] [['Y '1]])              ⇒  [['X '2] ['Y '1]]
(BIND '[X] ''[2] [['Y '1]])             ⇒  [['X ''2] ['Y '1]]
(BIND '[X Z] '[2 3] [['Y '1]])          ⇒  [['X '2] ['Z '3] ['Y '1]]
(BIND '[X Y] '[[2] 3] [['Y '1]])        ⇒  [['X '[2]] ['Y '3] ['Y '1]]
(BIND '[X [Z]] '[2 [3]] [['Y '1]])      ⇒  [['X '2] ['Z '3] ['Y '1]]
(BIND '[X] '2 [['Y '1]])                ⇒  {ERROR}
(BIND '[X] '''[2] [['Y '1]])            ⇒  {ERROR}
(BIND '(A . B) '(1 . 2) [['Y '1]])      ⇒  {ERROR}
```

(LET *[[P_1 E_1] ... [P_k E_k]]* <u>*BODY*</u>)

Designates the designation that *BODY* has in an environment which is like the current environment except extended by matching the patterns P_i to the results of normalizing the expressions E_i in environment the current environment. In other words all of the E_i are normalized in the same environment. It can be determined (because of the way in which rails are normalized) that the E_i will be normalized sequentially, but it is considered bad programming practice to depend on this fact (only BLOCK should be used for explicit sequential processing).

Properties: Kernel; env; abnormal.

Macro:
```
(LET [[P1 E1] ... [Pk Ek]] BODY)
≡> ((LAMBDA SIMPLE [P1 ... Pk] BODY) E1 ... Ek)
```

Examples:
```
(LET [[X 3] [Y 4]] (+ X Y))                 ⇒  7
(LET [[[A B] (REST [1 2 3])]] (+ A B))      ⇒  5
(LET [[X 3]]
    (LET [[X 4] [Y X]] (+ X Y)))            ⇒  7
```

(LETSEQ *[[P_1 E_1] ... [P_k E_k]] BODY*)

LETSEQ is like LET except that each expression E_{i+1} is normalized in the environment that results from extending the previous environment with the results of matching pattern P_i against the normalization of E_i.

Properties: Env; abnormal.

Macro:

```
(LETSEQ [[P1 E1][P2 E2] ... [Pk Ek]] BODY)
≡> (LET [[P1 E1]]
        (LETSEQ [[P2 E2] ... [Pk Ek]] BODY))
```

Examples:

```
(LET [[X 3]]
     (LETSEQ [[X 4] [Y X]] (+ X Y))) ⇒    8
```

(LETREC *[[V_1 E_1] ... [V_k E_k]] BODY*)

Like LET and LETSEQ except that each expression E_i is normalized in the environment that results from extending the original environment with the results of binding all of the variables V_i against the normalizations of their E_i.

Properties: Env; abnormal.

Macro:

```
(LETREC [[V1 E1][V2 E2] ... [Vk Ek]] BODY)
≡> (LET [[V1 'HUCAIRZ][P2 'HUCAIRZ] ... [Vk 'HUCAIRZ]]
        (BLOCK (SET V1 E1)
               (SET V2 E2)
               ...
               (SET Vk Ek)
               BODY))
```

Examples:

```
(LETREC [[EVEN (LAMBDA SIMPLE [N]
                 (IF (= N 0) $T (ODD (1- N))))]
         [ODD (LAMBDA SIMPLE [N]
                 (IF (= N 0) $F (NOT (EVEN N))))]]
   (SCONS (EVEN 2) (ODD 2)))          ⇒    [$T $F]
```

4.c.9. CONTROL

(EF *PREM C_1 C_2*)
(IF *PREM $\underline{C_1}$ $\underline{C_2}$*)

Both (IF *PREM* C_1 C_2) and (EF *PREM* C_1 C_2) designate the referent of C_1 or C_2 depending on whether *PREM* designates true or false, respectively. In the case of IF, C_1 (C_2) is normalized only if PREM designates true (false), whereas EF is fully (procedurally) extensional.

Φ-Type: **[TRUTH-VALUES × OBJECTS × OBJECTS] → OBJECTS**

Properties: Primitive (EF only); kernel; abnormal (IF only).

Examples:

```
1> (IF (= 1 1) 'A 'B)
1= 'A
1> (IF (= 1 2) 'A 'B)
1= 'B
1> (EF (= 1 2)
       (PRINT-STRING "Hello" PRIMARY-STREAM)
       (PRINT-STRING "Good-bye" PRIMARY-STREAM)) Hello Good-bye
1= 'OK
1> (IF (= 1 2)
       (PRINT-STRING "Hello" PRIMARY-STREAM)
       (PRINT-STRING "Good-bye" PRIMARY-STREAM)) Good-bye
1= 'OK
1> (EF [] 'A 'B)
ERROR: Truth value expected.
```

(COND *[P_1 $\underline{C_1}$]* ... *[P_k $\underline{C_k}$]*)

Designates C_i for the least i such that P_i designates true. Only P_1, P_2, ..., P_i and C_i are normalized. Error if no P_i designates true, or some P_i doesn't designate a truth value.

Properties: Kernel; abnormal.

Examples:

```
1> (COND [(= 1 2) 10]
         [(= 1 3) 20]
         [(= 1 1) 30]
         [$T 40])
1= '30
1> (COND [(= 1 2) (PRINT '10 PRIMARY-STREAM)]
         [(= 1 3) (PRINT '20 PRIMARY-STREAM)]
         [(= 1 1) (PRINT '30 PRIMARY-STREAM)]
         [$T      (PRINT '40 PRIMARY-STREAM)]) 30
1= 'OK
```

(BLOCK *C_1 ... $\underline{C_k}$*)

The results of normalizing C_1 through C_{k-1} are discarded, and the result of normalizing C_k is returned. Note that C_k is normalized tail-recursively with respect to the BLOCK.

Φ-Type: **[{OBJECTS}* × OBJECTS] → OBJECTS**

Examples:

```
1> (BLOCK 1 2 3)
1= 3
1> (BLOCK (PRINT-STRING "2 " PRIMARY-STREAM)
          (PRINT-STRING "+ " PRIMARY-STREAM)
          (PRINT-STRING "2 " PRIMARY-STREAM)
          'DONE) 2 + 2
1= 'DONE
```

(CATCH *C*)

Declaratively speaking, CATCH designates the identity function — it returns what *C* normalizes to. However, if (THROW *E*) is normalized in the process of normalizing *C* (and assuming that there are no intervening CATCHes) the result of normalizing *E* is immediately returned as the result of the enclosing (CATCH C).

Φ-Type: **[OBJECTS] → OBJECTS** *Properties:* (Hairy).

Examples:
```
1> (CATCH (+ 2 2))
1= 4
1> (CATCH (+ 2 (THROW 3)))
1= 3
1> (CATCH
     (BLOCK (THROW (+ 3 3))
            100))
1= 6
```

(THROW *C*)

Causes the result of normalizing C to be returned immediately as the result of the most recently executed enclosing CATCH. The current reflective level is abandoned if there is no enclosing CATCH.

Properties: (Hairy).

Examples:
```
1> (CATCH (BLOCK (PRINT-STRING "-2 " PRIMARY-STREAM)
                 (PRINT-STRING "-1 " PRIMARY-STREAM)
                 (THROW 'BLAST-OFF)
                 (PRINT-STRING "1 " PRIMARY-STREAM)
                 (PRINT-STRING "2 " PRIMARY-STREAM))) -2 -1
1= 'BLAST-OFF
1> (CATCH (+ (CATCH (* 5 3))
             (THROW (* 6 (THROW 4)))))
1= 4
1> (THROW (+ 2 2))
2= '4
```

(DELAY *C*)

Defers the normalization of C by embedding it in a LAMBDA expression.

Properties: Abnormal.

Macro: `(DELAY C)  ≡>  (LAMBDA SIMPLE [] C)`

Examples:
```
1> (SET X (DELAY (* Y Y)))
1= 'OK
1> (SET Y 7)
1= 'OK
1> (FORCE X)
1= 49
1> (SET Y 9)
1= 'OK
1> (FORCE X)
1= 81
1> (DEFINE NEW-IF
     (LAMBDA MACRO [P C1 C2]
       `(FORCE (EF ,P (DELAY ,C1) (DELAY ,C2)))))
1= 'NEW-IF
1> (NEW-IF (= (+ 2 2) 4)
           (PRINT 'YES PRIMARY-STREAM)
           (PRINT 'NO PRIMARY-STREAM))
YES
1= 'OK
```

(FORCE *C*)

Causes the normalization of the DELAYed expression designated by C.

Examples:
```
1> (SET X (DELAY (PRINT-STRING GREETING PRIMARY-STREAM)))
1= 'OK
1> (SET GREETING "Hi there")
1= 'OK
1> (FORCE X) Hi there
1= 'OK
1> (SET GREETING "Good-bye")
1= 'OK
1> (FORCE X) Good-bye
1= 'OK
```

(SELECT *INDEX [M_1 $\underline{C_1}$] ... [M_k $\underline{C_k}$]*)
(SELECTQ *INDEX [M_1 $\underline{C_1}$] ... [M_k $\underline{C_k}$]*)

SELECTQ allows one of several clauses (the C_i) to the processed based upon the designation of INDEX. The M_i are tested from left to right, stopping as soon as a clause is selected. If M_i is an atom, the ith clause will be selected if the selector designates this atom; if M_i is a rail, the ith clause will be selected if the selector is a member of this rail; otherwise, M_i should be the boolean $T which will always be selected, if given half a chance. Error if no clause is selected. SELECT is similar except that the selection is based on the designation of M_i instead of the unnormalized structure.

Properties: Abnormal.

Macro: E.g.
```
(SELECTQ INDEX
   [A C1]
   [[A1 ... AN] C2]
   ...
   [$T Ck])
≡>
(LET [[{selector} INDEX]]
   (COND [(= {selector} 'A) C1]
         [(MEMBER {selector} '[A1 ... AN]) C2]
         ...
         [$T Ck]))
```

Example:
```
1> (DEFINE ACTIVITY
      (LAMBDA SIMPLE [DAY]
         (SELECTQ DAY
            [SUNDAY 'SLEEP]
            [[MONDAY THURSDAY] 'WORK]
            [$T 'RUMINATE])))
1= 'ACTIVITY
1> (ACTIVITY 'SUNDAY)
1= 'SLEEP
1> (DEFINE ACTIVITY-2
      (LAMBDA SIMPLE [DAY]
         (SELECT DAY
            ['SUNDAY 'SLEEP]
            [['MONDAY 'THURSDAY] 'WORK]
            [$T 'RUMINATE])))
1= 'ACTIVITY-2
1> (ACTIVITY-2 'THURSDAY)
1= 'WORK
```

**(DO *[[VAR_1 $INIT_1$ $NEXT_1$] ... [VAR_k $INIT_k$ $NEXT_k$]]*
[[EXIT-TEST$_1$ $\underline{RETURN_1}$] ... [EXIT-TEST$_j$ $\underline{RETURN_j}$]]
BODY)**

DO is a general-purpose iteration operator (taken from SCHEME, and generalized from MACLISP and ZETALISP). The variables VAR_1 through VAR_k are initially bound to the results of normalizing the expressions $INIT_1$ through $INIT_k$ (these "initializing" expressions are normalized sequentially, but all of them are normalized before any of the bindings are established). Then each of the EXIT-TEST$_i$ are processed in order; if any is true, the corresponding expression $RETURN_i$ is processed, with the result of that $RETURN_i$ being returned as the result of the entire DO form. If none of the tests are true, BODY is processed (result ignored), and the variables VAR_1 through VAR_k are bound to the results of processing $NEXT_1$ through $NEXT_k$, and the process repeats. The $NEXT_i$ are normalized in an environment in which all of the VAR_i remain bound to their previous bindings. BODY may be omitted.

Properties: Abnormal; env.

Macro:

```
(DO [[VAR1 INIT1 NEXT1] ... [VARk INITk NEXTk]]
    [[EXIT-TEST1 RETURN1] ... [EXIT-TESTj RETURNj]]
    BODY)
≡>
(LETREC
    [[{loop}
        (LAMBDA SIMPLE [VAR1 ... VARk]
           (COND [EXIT-TEST1 RETURN1]
                 ...
                 [EXIT-TESTj RETURNj]
                 [$T (BLOCK BODY ({loop} NEXT1 ... NEXTk))]))]]
    ({loop} INIT1 ... INITk))
```

Example:

```
1> (DEFINE NEW-REVERSE
      (LAMBDA SIMPLE [VEC]
         (DO [[V VEC (REST V)]
              [R ((VECTOR-CONSTRUCTOR VEC)) (PREP (1ST V) R)]]
             [[(EMPTY V) R]])))
1= 'NEW-REVERSE
1> (NEW-REVERSE "Rogatien")
1= "neitagoR"
```

4.c.10. TRUTH VALUE OPERATIONS

(NOT *E*)

True if *E* designates false, and false if *E* designates true.

Φ-Type: [TRUTH-VALUES] → TRUTH-VALUES

Examples:

```
(NOT $F)           ⇒  $T
(NOT (EVEN 102))   ⇒  $F
(NOT 1)            ⇒  {ERROR: Truth value expected.}
```

(AND E_1 E_2 ... E_k)
(OR E_1 E_2 ... E_k)

(AND E_1 E_2 ... E_k) is true just in case all the E_i are true; (OR E_1 E_2 ... E_k) is true just in case at least one of the E_i is true. Procedurally, these forms normalize their arguments one-by-one only until a deciding case is found ($F for AND; $T for OR); thus they may be able to return even if some of their arguments are non-terminating. *k* may be 0; (AND) returns **$T**; (OR) returns $F.

Φ-Type: [{TRUTH-VALUES}*] → TRUTH-VALUES *Properties:* Kernel (AND only); abnormal.

Examples:

```
(AND (= 1 1) (= 1 2))                  ⇒  $F
(OR (= 1 0) (= 1 2) (= 1 1))           ⇒  $T
(AND)                                  ⇒  $T
(OR)                                   ⇒  $F
(LET [[X 3]]
   (BLOCK (AND (= 1 2)
               (BLOCK (SET X 4) $T))
          X))                          ⇒  3
```

4.c.11. STRUCTURAL SIDE EFFECTS

(REPLACE S_1 S_2)

Replaces the pair, rail, atom, or closure designated by S_1 with the structure of the same type designated by S_2. Returns 'OK (therefore it will typically be used only within the scope of a BLOCK); however, subsequent to its execution the field will be altered in such a way that *every* relationship in which the designation of S_1 participated will be changed to have the designation of s_2 as its participant (with the consequence that the designation of s_1 becomes henceforth inaccessible). REPLACE is not defined over the other internal structure types: numerals, charats, streamers, or handles. REPLACE is a very dangerous operation that should be used with extreme caution.

Φ-Types: *[PAIRS × PAIRS] → ATOMS* *Properties:* Primitive; smash.
[RAILS × RAILS] → ATOMS
[CLOSURES × CLOSURES] → ATOMS
[ATOMS × ATOMS] → ATOMS

Examples:
```
(LET [[X '(+ 2 3)]]
   (BLOCK (REPLACE (CDR X) '[20 30])
          X))                              ⇒  '(+ 20 30)
(LET [[X '[]]]
   (BLOCK (REPLACE X '[NEW TAIL])
          X))                              ⇒  '[NEW TAIL]
(LET [[X '[A1 A2]]]
   (BLOCK (REPLACE 'A1 'A2 )
          X))                              ⇒  '[A2 A2]
```

(RPLACA *PAIR NEW-CAR*)
(RPLACD *PAIR NEW-CDR*)

RPLACA (RPLACD) alters the pair designated by *PAIR*, making its CAR (CDR) be the internal structure designated by *NEW-CAR* (*NEW-CDR*). Returns 'OK.

Φ-Types: *[PAIRS × STRUCTURES] → ATOMS* *Properties:* Smash.

Examples:
```
1> (SET X '(A . B))
1= 'OK
1> (SET Y X)
1= 'OK
1> (RPLACA X 'C)
1= 'OK
1> X
1= '(C . B)
1> Y
1= '(C . B)
```

(RPLACN *N RAIL NEW-ELEMENT*)

RPLACN alters the rail designated by *RAIL*, making its *N*th component be the internal structure designated by *NEW-ELEMENT*. 'OK is returned.

Φ-Types: *[NUMBERS × RAILS × STRUCTURES] → ATOMS* *Properties:* Smash.

Examples:
```
1> (SET X '[ONE TWO THREE])
1= 'OK
1> (SET Y (REST X))
1= 'OK
1> (RPLACN 2 X '**)
1= 'OK
1> X
1= '[ONE ** THREE]
1> Y
1= '[** THREE]
```

(RPLACT *N RAIL NEW-TAIL*)

(RPLACT *N RAIL NEW-TAIL*) replaces (using REPLACE) the *N*th TAIL of the rail designated by *RAIL* with the rail designated by *NEW-TAIL*. Returns 'OK.

Φ-Types: *[NUMBERS × RAILS × RAILS] → ATOMS* *Properties:* Smash.

Examples:
```
1> (SET X '[ONE TWO THREE])
1= 'OK
1> (SET Y (REST X))
1= 'OK
1> (SET Z (REST Y))
1= 'OK
```

```
1> (RPLACT 1 X '[END])
1= 'OK
1> X
1= '[ONE END]
1> Y
1= '[END]
1> Z
1= '[THREE]
```

4.c.12. LEVEL CROSSING OPERATORS

(UP *S*)
↑S

Designates the form to which *S* normalizes. '↑*S*' expands to (UP *S*) in the standard notation. Note that UP, although it is not a reflective procedure, is nonetheless not strictly extensional, since what it designates is a function not only of its arguments' *designation*, but also of its argument's *procedural consequence* (what it returns).

Φ-Type: *[OBJECTS] → STRUCTURES* *Properties:* Primitive; kernel.

Examples:

```
↑5                          ⇒  '5
↑(+ 2 3)                    ⇒  '5
↑(LAMBDA SIMPLE [X] X)      ⇒  '{closure: SIMPLE {global} [X] X}
['(= 2 3) ↑(= 2 3)]         ⇒  ['(= 2 3) '$F]
(LET [[X [2 3]]]
     (= X [2 3]))           ⇒  $T
(LET [[X [2 3]]]
     (= ↑X ↑[2 3]))         ⇒  $F
```

(DOWN *S*)
↓S

If *S* designates R — a normal-form designator — then (DOWN *EXP*) will normalize to R. '↓*S*' expands to '(DOWN *S*)' in the standard notation.

Φ-Type: *[STRUCTURES] → OBJECTS* *Properties:* Primitive; kernel.

Examples:

```
↓'4                         ⇒  4
↓(NTH 2 '[10 20 30])        ⇒  20
↓3                          ⇒  {ERROR: Structure expected.}
↓↑$T                        ⇒  $T
↓'X                         ⇒  {ERROR: Not a normal form structure.}
```

(REFERENT *EXP ENV*)

If *EXP* designates R and R normalizes to R' in the environment designated by *ENV*, then (REFERENT *EXP ENV*) will return R'. Thus REFERENT can obtain the referent of any structure, whereas DOWN is restructed to normal-form structures.

Φ-Type: *[STRUCTURES × SEQUENCES] → OBJECTS*

Properties: (Arbitrary effects due to sub-normalization).

Examples:

```
(REFERENT '1 GLOBAL)                    ⇒  1
(REFERENT 'X [['X '1]])                 ⇒  1
(REFERENT ''(+ 2 2) [])                 ⇒  '(+ 2 2)
(REFERENT (PCONS '+ '[2 2]) GLOBAL)     ⇒  4
```

4.c.13. SYSTEM UTILITIES

(VERSION)

Designates a character string that identifies the 3-LISP implementation.

Φ-Type: *[ ] → SEQUENCES*

Example:
```
1> (VERSION)
1= "3-LISP version A00. May 1, 1983"
```

(LOADFILE *FILENAME*)

Loads 3-LISP definitions from the file with the same spelling as the atom designated by *FILENAME*. These definitions, which are stored as character strings, are stuffed into the primary stream so that subsequent READs will see them. Returns 'OK. (This is an interim mechanism; work is under way in providing a more reasonable means of saving and loading input files.)

Φ-Type: *[ATOMS] → ATOMS* *Properties:* Primitive; I/O.

Example:
```
1> (LOADFILE 'MY-FILE)
1= 'OK
```
(... contents of file MY-FILE are read in at this point.)

(LOAD *FILENAME*)

A variant of LOADFILE that does not normalize its argument.

Properties: Abnormal; I/O.

Macro: (LOAD *FILENAME*) ≡> (LOADFILE '*FILENAME*)

Example:
```
1> (LOAD MY-FILE)
1= 'OK
```
(... contents of file MY-FILE are read in at this point.)

(EDITDEF *PROCNAME*)

Every time a character string of the form '(DEFINE *FOO FUM*)' or '(SET *FOO FUM*)' are encountered by READ, the string is remembered with the atom *FOO*. Anytime later, (EDITDEF '*FOO*) will retrieve this string so that it can be edited (with INTERLISP-D's TTYIN). Upon completion of editing, the string is queued for READ, just as is done when a file is LOADed. Returns 'OK. Note that the code for the standard procedures can be accessed in this manner. (This too is an interim mechanism; work is under way in providing a more reasonable means of editing 3-LISP code.)

Φ-Type: *[ATOMS] → ATOMS* *Properties:* Primitive; I/O.

Example:
```
1> (EDITDEF 'FOO)
```
(... the text string definition of FOO is displayed for editing.)

(EDIT *PROCNAME*)

A variant of EDITDEF that does not normalize its argument.

Properties: Abnormal; I/O.

Macro: (EDIT *PROCNAME*) ≡> (EDITDEF '*PROCNAME*)

Example:
```
1> (EDIT NORMALISE)
```
(... the text string definition of NORMALIZE is displayed for editing.)

4.c.14. INPUT and OUTPUT

PRIMARY-STREAM

Designates the primary input-output stream through which all communication is done. Note that only characters can be read from or written to this stream.

Φ-Type: ***STREAMS*** *Properties:* Variable.

Examples:
```
PRIMARY-STREAM              ⇒   {streamer}
(TYPE PRIMARY-STREAM)       ⇒   'STREAM
```

(INPUT *STREAM*)

Designates the next item in the stream designated by *S*. It should be assumed that *S* is side-effected by this operation.

Φ-Type: ***[STREAMS] → OBJECTS*** *Properties:* Primitive; I/O.

Examples:
```
1> (INPUT PRIMARY-STREAM) ?
1= #?
1> [(INPUT PRIMARY-STREAM) (INPUT PRIMARY-STREAM)] Oz
1= "Oz"
```

(OUTPUT *S STREAM*)

Puts the structure designated by *S* into the stream designated by *STREAM*. Returns 'OK. It should be assumed that *STREAM* is side-effected by this operation.

Φ-Type: ***[OBJECTS × STREAMS] → ATOMS*** *Properties:* Primitive; I/O.

Examples:
```
1> (OUTPUT #? PRIMARY-STREAM) ?
1= 'OK
1> [(OUTPUT #O PRIMARY-STREAM) (OUTPUT #z PRIMARY-STREAM)] Oz
1= 'OK
```

(NEWLINE *STREAM*)

Outputs a carriage return character to the stream designated by *STREAM*. Returns 'OK.

Φ-Type: ***[STREAMS] → ATOMS*** *Properties:* I/O.

Examples:
```
1> (BLOCK (NEWLINE PRIMARY-STREAM)
          (OUTPUT #? PRIMARY-STREAM))
?
1= 'OK
```

(PROMPT&READ *N STREAM*)

Outputs a level *N* input prompt to the stream designated by *STREAM*, READs an expression from that stream, and returns a designator of that expression.

Φ-Type: ***[NUMBERS × STREAMS] → STRUCTURES*** *Properties:* I/O.

Examples:
```
1> (PROMPT&READ 100 PRIMARY-STREAM)
100> Hello
1= 'HELLO
```

(PROMPT&REPLY *ANSWER N STREAM*)

PRINTs the structure designated by *ANSWER*, preceded by a level *N* output prompt, to the stream designated by *STREAM*. Returns 'OK.

Φ-Type: ***[STRUCTURES × NUMBERS × STREAMS] → ATOMS*** *Properties:* I/O.

Examples:
```
1> (PROMPT&REPLY 'HELLO 100 PRIMARY-STREAM)
100= HELLO
1= 'OK
1> (PROMPT&REPLY (PROMPT&READ 15 PRIMARY-STREAM) 15 PRIMARY-STREAM))
15> (+ 2 2)
15= (+ 2 2)
1= 'OK
```

(PRINT-STRING *STRING STREAM*)

OUTPUTS the character in the string designated by *STRING* to the stream designated by *STREAM*. Returns 'OK.

Φ-Type: *[SEQUENCES × STREAMS] → ATOMS* *Properties:* I/O.

Examples:
```
1> (PRINT-STRING "Hello there" PRIMARY-STREAM) Hello there
1= 'OK
```

(READ *STREAM*)

READ parses and internalizes a character sequence notating a 3-LISP structure and returns a handle to that structure. The sequence of characters is obtained from the stream designated by *STREAM*. Note that all pairs and rails accessible from the result were previously completely inaccessible.

Φ-Type: *[STREAMS] → STRUCTURES* *Properties:* I/O; cons. (*Not currently explained.*)

Examples:
```
1> (READ PRIMARY-STREAM) (A . B)
1= '(A . B)
1> (READ PRIMARY-STREAM) '$T
1= ''$T
```

(PRINT *S STREAM*)

PRINT externalizes the structures designated by *S* and sends the sequence of character to the stream designated by *STREAM*. Returns 'OK.

Φ-Type: *[STRUCTURES × STREAMS] → ATOMS* *Properties:* I/O; cons. (*Not currently explained.*)

Examples:
```
1> (PRINT '(A . B) PRIMARY-STREAM) (A . B)
1= 'OK
1> (PRINT ''$T PRIMARY-STREAM) '$T
1= 'OK
```

(INTERNALIZE *STRING*)

STRING is taken as designating a character sequence that notates some 3-LISP structure. INTERNALIZE returns a handle to this structure. Note that all pairs and rails accessible from the result were previously completely inaccessible.

Φ-Type: *[SEQUENCES] → STRUCTURES* *Properties:* Cons. (*Not currently implemented.*)

Examples:
```
1> (INTERNALIZE "(A . B)")
1= '(A . B)
1> (INTERNALIZE (PREP #' "$T"))
1= ''$T
```

(EXTERNALIZE *S*)

The internal structure designated by *S* is converted to a character string that would notate this structure (up to structure isomophism). The result designates this character sequence. Note that some structures, such as circular rails, will usually cause this procedure to loop indefinitely.

Φ-Type: *[STRUCTURES] → SEQUENCES* *Properties:* Cons. (*Not currently implemented.*)

Examples:
```
1> (EXTERNALIZE '(A . B))
1= "(A . B)"
1> (2ND (EXTERNALIZE ''$T))
1= "$"
```

4.c.15. OTHER GENERAL UTILITIES

(ID *E*)

ID designates the single argument identity function. (ID E) returns what E normalizes to.

Φ-Type: *[OBJECTS] → OBJECTS*

Examples:

```
(ID 3)                    ⇒  3
(ID (+ 2 2))              ⇒  4
(ID '(+ 2 3))             ⇒  '(+ 2 3)
(ID ID)                   ⇒  {simple ID closure}
```

(ID* *E₁ E₂ ... E_K*)

ID* designates the multi-argument identity function. (ID* . E) returns what E normalizes to.

Φ-Type: *OBJECTS → OBJECTS*

Examples:

```
(ID* 3)                   ⇒  [3]
(ID* (+ 2 2) (TYPE '1))   ⇒  [4 'NUMERAL]
(ID* '(+ 2 3))            ⇒  ['(+ 2 3)]
(ID* . GLOBAL)            ⇒  {global}
(ID* . (+ 2 2))           ⇒  4
```

(MACRO-EXPANDER *FUN*)

FUN must normalize to a closure that was generated with MACRO. Designates a function that will perform the macro expansion entailed in normalizing a call to *FUN*.

Φ-Type: *[FUNCTIONS] → FUNCTIONS*

Examples:

```
((MACRO-EXPANDER DELAY)
    '[(FOO X)])           ⇒  '(LAMBDA SIMPLE [] (FOO X))
((MACRO-EXPANDER LET)
    '[[[X 1] (+ X 2)]])   ⇒  '((LAMBDA SIMPLE [X] (+ X 2)) 1)
```

(QUOTE *EXP*)

Returns a designator of the structure *EXP*. Note that QUOTE doesn't normalize its argument. (It is interesting to see what happens when QUOTE is used as a functional argument; other than that, QUOTE is never really needed since the 3-LISP structural field provides this capability via handles.)

Properties: Abnormal.

Examples:

```
(QUOTE 2)                 ⇒  '2
'2                        ⇒  '2
(QUOTE (+ 2 2))           ⇒  '(+ 2 2)
'(+ 2 2)                  ⇒  '(+ 2 2)
(NORMALISE 'A [] QUOTE)   ⇒  '(BINDING EXP ENV)
(MAP QUOTE [1 2])         ⇒  ['(1ST (2ND ARGS)) '(1ST (2ND
ARGS))]
```

4.c.16. PROCESSOR

(NORMALIZE *EXP ENV CONT*)

Normalizes the structure designated by *EXP* in the environment designated by *ENV* with continuation designated by *CONT*. Under normal circumstances, the normal-form designator that results from this normalization will be passed as the single argument to the continuation. Error if *EXP* does not designate a structure.

Φ-Type: *[STRUCTURES × SEQUENCES × FUNCTIONS] → OBJECTS* *Properties:* Kernel; CPS.

Examples:
```
(NORMALIZE '1 [] ID)                ⇒ '1
(NORMALIZE 'X [['X '1]] ID)         ⇒ '1
(NORMALIZE '(+ 2 2) GLOBAL ID)      ⇒ '4
(NORMALIZE '+ GLOBAL QUOTE)         ⇒ '(BINDING EXP ENV)
(NORMALIZE '$T GLOBAL QUOTE)        ⇒ 'EXP
```

(REDUCE *PROC ARGS ENV CONT*)

Reduces the referent of the structure designated by *PROC* with the referent of the structure designated by *ARGS* in the environment designated by *ENV* with continuation designated by *CONT*. Under normal circumstances, the normal-form designator that results from this process will be passed as the single argument to the continuation.

Φ-Type: *[STRUCTURES × STRUCTURES × SEQUENCES × FUNCTIONS] → OBJECTS*

Properties: Kernel; CPS.

Examples:
```
(REDUCE '+ '[2 2] GLOBAL ID)        ⇒ '4
(REDUCE 'IF '[$T 1 2] GLOBAL ID)    ⇒ '1
(REDUCE '+ '[2 2] GLOBAL
   (LAMBDA MACRO [X] ↑X))           ⇒ '↑(↓PROC! . ↓ARGS!)
```

(NORMALIZE-RAIL *RAIL ENV CONT*)

Normalizes the rail designated by *RAIL* in the environment designated by *ENV* with continuation designated by *CONT*. Under normal circumstances, the normal form rail that results from this processing will be passed as the single argument to the continuation.

Φ-Type: *[RAILS × SEQUENCES × FUNCTIONS] → OBJECTS* *Properties:* Kernel; CPS.

Examples:
```
(NORMALIZE-RAIL '[1] [] ID)                 ⇒ '[1]
(NORMALIZE-RAIL '[X X] [['X '1]] ID)        ⇒ '[1 1]
(NORMALIZE-RAIL '[(+ 2 2)] GLOBAL ID)       ⇒ '[4]
(NORMALIZE-RAIL '[+] GLOBAL
   (LAMBDA MACRO [X] ↑X))                   ⇒ '(PREP FIRST! REST!)
(NORMALIZE-RAIL '[] GLOBAL
   (LAMBDA MACRO [X] ↑X))                   ⇒ '(RCONS)
```

(READ-NORMALIZE-PRINT *LEVEL ENV STREAM*)

Starts a READ, NORMALIZE, PRINT loop with *ENV* designating the initial environment. *STREAM* designates the stream through which this driver loop communicates; the designation of *LEVEL* is used as a (hopefully unique) identifying prompt. Under normal circumstances, READ-NORMALIZE-PRINT will not terminate.

Φ-Type: *[OBJECTS × SEQUENCES × STREAMS] → OBJECTS* *Properties:* CPS.

Examples:
```
1> (READ-NORMALIZE-PRINT 'NEW GLOBAL PRIMARY-STREAM)
'NEW> (+ 2 2)
'NEW= 4
'NEW> ; This level is just as good as the old one.
```

(NORMAL *S*)

True just in case *S* designates a normal-form internal structure.

Φ-Type: ***[STRUCTURES] → TRUTH-VALUES*** *Properties:* Kernel.

Examples:
```
(NORMAL '3)          ⇒  $T
(NORMAL '(+ 2 3))    ⇒  $F
(NORMAL ↑(+ 2 3))    ⇒  $T
(NORMAL '[1 2 3])    ⇒  $T
(NORMAL '[1 2 A])    ⇒  $F
(NORMAL 'A)          ⇒  $F
(NORMAL ''A)         ⇒  $T
```

(NORMAL-RAIL *RAIL*)

True just in case *RAIL* designates a normal-form rail.

Φ-Type: ***[RAILS] → TRUTH-VALUES*** *Properties:* Kernel.

Examples:
```
(NORMAL-RAIL '[])          ⇒ $T
(NORMAL-RAIL '[1 $T #C])   ⇒ $T
(NORMAL-RAIL '[1 2 A])     ⇒ $F
```

(PRIMITIVE *CLOSURE*)

True just in case *CLOSURE* designates one of the thirty or so primitive closures; false otherwise.

Φ-Type: ***[CLOSURES] → TRUTH-VALUES*** *Properties:* Kernel.

Examples:
```
(PRIMITIVE ↑+)            ⇒  $T
(PRIMITIVE ↑NORMALISE)    ⇒  $F
(PRIMITIVE ↑IF)           ⇒  $F
```

PRIMITIVE-CLOSURES

This variable designates the sequence of primitive closures.

Φ-Type: ***SEQUENCES*** *Properties:* Variable; kernel.

Examples:
```
(MEMBER ↑EF PRIMITIVE-CLOSURES)    ⇒   $T
(MEMBER ↑IF PRIMITIVE-CLOSURES)    ⇒   $F
```

GLOBAL

This variable designates the global environment. The rail to which GLOBAL is bound is shared across all reflective levels, and is a tail of the environment designator captured in most closures.

Φ-Type: ***SEQUENCES*** *Properties:* Variable.

Examples:
```
1> (DEFINE LAST
     (LAMBDA SIMPLE [S]
       (IF (UNIT S) (1ST S) (LAST (REST S)))))
1= 'LAST
1> (SET XXXX '[HORTUS SICCUS])
1= 'OK
1> (BINDING 'XXXX GLOBAL)
1= ''[HORTUS SICCUS]
1> (LAST GLOBAL)
1= ['XXXX ''[HORTUS SISSUS]]
```

(COND-HELPER *ARGS ENV CONT*)
(BLOCK-HELPER *CLAUSES ENV CONT*)
(AND-HELPER *ARGS ENV CONT*)
(OR-HELPER *ARGS ENV CONT*)

These are auxiliary procedures used in the definition of COND, BLOCK, AND, and OR, respectively; e.g., COND is defined as (REFLECTIFY COND-HELPER).

Φ-Type: ***[STRUCTURES × SEQUENCES × FUNCTIONS] → OBJECTS***

Properties: Kernel (COND-HELPER and AND-HELPER only); CPS; (smash; cons; I/O).

Examples:

```
(COND-HELPER '[[(= 2 2) 1][$T 2]] GLOBAL ID)   ⇒  '2
(BLOCK-HELPER '[X X X] [['X '1]] ID)           ⇒  '1
(AND-HELPER '[(= 2 2) (= 3 3)] GLOBAL ID)      ⇒  '$T
(OR-HELPER '[(= 2 2) (= 3 4)] GLOBAL ID)       ⇒  '$T
```

5. Running 3-LISP

In comparison to the LISP MACHINE LISP implementation of 3-LISP presented in the appendix of [Smith 82a], the current implementation is a couple of orders of magnitude more efficient. This version of 3-LISP is implemented in INTERLISP-D for the Xerox 1100, 1108, and 1132 processors; this section endeavors to explain to someone familiar with INTERLISP-D how to go about starting up 3-LISP.

5.a. Starting Off

Restoring the 3-LISP SYSOUT file in the standard way will put you at the INTERLISP top level. After connecting to your directory, you invoke the function 3-LISP to get to the (level 1) 3-LISP top level. Important note: You *cannot* mix INTERLISP and 3-LISP code; i.e this is not an embedded implementation like, say, the original implementations of SCHEME.

5.b. Special Characters

In addition to the notational conventions explained in §3 the user must be aware of the following special interrupt characters.

Character	*In 3-LISP*	*In Interlisp*
D^c	Hard reset to 3-LISP '1>' top level.	Hard reset to INTERLISP top level.
Y^c	Exit to INTERLISP.	Enter 3-LISP.

As mentioned in §3, the backslash character '\' should be used in place of the down-arrow character '↓'. However, Xerox 1100 series keyboards do not have the back-quote character '`' — type the tilde character '~' instead.

5.c. Editing

The TTYIN package is used to read 3-LISP expressions, thereby providing parenthesis balancing and the usual stable of input editing capabilities (with the exception of automatic (re-)formatting, which does not work properly due to read macros).

Expressions of the from '(DEFINE *Foo Expression*)' or '(SET *Foo Expression*)' are treated specially by READ. When such an expression is encountered, it is saved in the INTERLISP world as the 3-LISP-FNS file package type definition of the literal atom named *Foo*. The entire text of the expression is saved exactly the way it was entered; a subsequent call to the 3-LISP primitive EDITDEF (or EDIT) redisplay the expression in the editor window and open it up for editing (again, with TTYIN). After the text in the window has been fixed, a single well-formed 3-LISP expression is queued for READ. The modified expression is not automatically written back; what happens will depend on whether the modified expression begins in 'DEFINE' or 'SET'. An editing session can be abandoned by using the D^c interrupt; the modified expression will be discarded.

Important note: Although the editor window can be enlarged, it cannot be scrolled. This imposes an unfortunate constraint on the length of one's 3-LISP procedure definitions.

5.d. Saving Your Work

An mentioned above, the INTERLISP file package type 3-LISP-FNS is used for recording the text string definitions of 3-LISP variables that acquire their binding via DEFINE or SET. These definitions can be assigned to files as per the normal INTERLISP mechanisms (e.g., CLEANUP, FILES?, etc.).

The 3-LISP primitive LOADFILE (or LOAD) is implemented with an INTERLISP LOAD of the named file. Once loaded, all 3-LISP-FNS contained on that file are extracted and queued for READ. (Note that it is necessary to connect to the appropriate directory prior to doing a 3-LISP LOADFILE since the file name cannot contain special characters like '{', '}', '.' or ';'.)

5.e. A Word on Protection

The current implementation of 3-LISP protects itself from accidental damage by disallowing REPLACE operations on all of the atoms, pairs, rails, and closures created as part of the standard system. The one exception, of course, is the foot of the global environment rail, which must be REPLACEable if global SETS and DEFINES are to be possible. However, the text string definitions of the standard procedures are *not protected* since they play no effectively connected role in the operation of the 3-LISP processor. Since it is convenient to be able to consult the standard definitions from time to time, and to clone them when a variant is required, it is best to avoid mangling them (i.e., always leave the editor via D^c to ensure that the modified definition is not saved).

References

Abelson, H., and Sussman, G. *Structure and Interpretation of Computer Programs*, M.I.T. Artificial Intelligence Laboratory Technical Report AI-TR-735 (July 1983).

Allen, J. *Anatomy of LISP.* New York: McGraw-Hill (1978).

Galley, S., and Pfister, G. *The MDL Language.* Programming Technology Division Document SYS.11.01, Laboratory of Computer Science, M.I.T. (1975).

Gordon, M. J. C., *The Denotational Description of Programming Languages: An Introduction*, New York: Springer-Verlag (1979).

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

Moon, D. "MacLisp Reference Manual", M.I.T. Laboratory for Computer Science, Cambridge, Mass. (1974).

Pitman, K. "Special Forms in LISP", *Conference Record of the 1980* LISP *Conference*, Stanford University (August 1980), pp. 179-187.

Rees, J.A., and Adams, N.I. IV, "T: A Dialect of LISP", *Conference Record of the 1982* LISP *and Functional Programming Conference*, Pittsburgh, Pensylvania (August 1982), pp. 114-122.

Sannella, M. *INTERLISP Reference Manual*, Pasadena: Xerox Special Information Systems (October 1983).

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

Smith. B. "Linguistic and Computational Semantics", ACL Conference (1982b).

Smith, B. "Reflection and Semantics in Lisp", Proceedings 1984 ACM Principles of Programming Languages Conference, Salt Lake City, Utah, pp. 23-35 (1984a).

Smith, B., and des Rivières, J. "Interim 3-LISP Reference Manual", Palo Alto: Xerox PARC ISL Technical Report (1984b).

Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, Cambridge: MIT Press (1977).

Sussman, G., and Steele, G. "SCHEME: An Interpreter for Extended Lambda Calculus", M.I.T. Artificial Intelligence Laboratory Memo AIM-349 (1975).

Steele, G., and Sussman, G. "LAMBDA: The Ultimate Imperative", M.I.T Artificial Intelligence Laboratory Memo AIM-353 (1976a).

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

Steele, G. "Debunking the "Expensive Procedure Call" Myth", M.I.T. Artificial Intelligence Laboratory Memo AIM-443 (1977).

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).

Steele, G., "An Overview of Common LISP", *Conference Record of the 1982* LISP *and Functional Programming Conference*, Pittsburgh, Pensylvania (August 1982), pp. 98-107.

Teitelman, W., *INTERLISP Reference Manual*. Xerox Palo Alto Research Center, Palo Alto, Calif. (October 1978).

Weinreb, D., and Moon, D. *LISP Machine Manual*, Cambridge: Massachusetts Institute of Technology (July 1981).

White, Jon L. "NIL: A Perspective", *Proceedings of the 1979* MACSYMA *Users' Conference*, Washington, D.C. (June 1979), pp. 190-199.

Appendix A. Standard Procedure Definitions

This appendix contains definitions for all of the standard procedures described in §4, and illustrates the structure of the primitive closures. Some of the definitions given here (such as for LAMBDA and DEFINE) are viciously circular, in that they use themselves (the definition of DEFINE, for example, starts out as (define DEFINE ...), but these circular definitions are far more illuminating than the code that is actually used to construct the appropriate closures. What is true about these definitions is that once the procedures are defined, the definitions presented here will leave them semantically unchanged.

The Reflective Processor (the "Magnificent Seven")

```
(define READ-NORMALIZE-PRINT
   (lambda simple [level env stream]
      (normalize (prompt&read level stream) env
         (lambda simple [result]                          ; Continuation C-REPLY
            (block (prompt&reply result level stream)
                   (read-normalize-print level env stream))))))

(define NORMALIZE
   (lambda simple [exp env cont]
      (cond [(normal exp) (cont exp)]
            [(atom exp) (cont (binding exp env))]
            [(rail exp) (normalize-rail exp env cont)]
            [(pair exp) (reduce (car exp) (cdr exp) env cont)])))

(define REDUCE
   (lambda simple [proc args env cont]
      (normalize proc env
         (lambda simple [proc!]                           ; Continuation C-PROC!
            (if (reflective proc!)
                (↓(de-reflect proc!) args env cont)
                (normalize args env
                   (lambda simple [args!]                 ; Continuation C-ARGS!
                      (if (primitive proc!)
                          (cont ↑(↓proc! . ↓args!))
                          (normalize (body proc!)
                                     (bind (pattern proc!) args! (environment proc!))
                                     cont)))))))))

(define NORMALIZE-RAIL
   (lambda simple [rail env cont]
      (if (empty rail)
          (cont (rcons))
          (normalize (1st rail) env
             (lambda simple [first!]                      ; Continuation C-FIRST!
                (normalize-rail (rest rail) env
                   (lambda simple [rest!]                 ; Continuation C-REST!
                      (cont (prep first! rest!)))))))))
```

Processor Utilities

```
(define NORMAL
   (lambda simple [x]
      (let [[tx (type x)]]
         (cond [(member tx ['atom 'pair]) $F]
               [(member tx ['numeral 'charat 'boolean 'handle
                            'closure 'streamer]) $T]
               [(= tx 'rail) (normal-rail x)]))))

(define NORMAL-RAIL
   (lambda simple [rail]
      (cond [(empty rail) $T]
            [(normal (1st rail)) (normal-rail (rest rail))]
            [$T $F])))

(define PRIMITIVE
   (lambda simple [closure]
      (member closure primitive-closures)))

(set PRIMITIVE-CLOSURES
   [↑+ ↑- ↑* ↑/ ↑< ↑> ↑<= ↑>= ↑ef ↑type ↑replace
    ↑nth ↑empty ↑tail ↑length ↑rcons ↑prep ↑scons
    ↑ccons ↑procedure-type ↑environment-designator
    ↑pattern ↑body ↑pcons ↑car ↑cdr ↑acons ↑up ↑down
    ↑input ↑output ↑loadfile ↑editdef])

(define BINDING
   (lambda simple [var env]
      (if (= var (1st (1st env)))
          (2nd (1st env))
          (binding var (rest env)))))

(define BIND
   (lambda simple [pattern args bindings]
      (cond [(atom pattern) (prep [pattern args] bindings)]
            [(handle args) (bind pattern (map up ↓args) bindings)]
            [(and (empty pattern) (empty args)) bindings]
            [$T (bind (1st pattern)
                      (1st args)
                      (bind (rest pattern) (rest args) bindings))])))

(define REFLECTIVE
   (lambda simple [closure]
      (= (procedure-type closure) 'reflect)))

(define DE-REFLECT
   (lambda simple [closure]
      (ccons 'simple
             (environment-designator closure)
             (pattern closure)
             (body closure))))
```

Naming and Procedure Definition

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

(define SIMPLE
   (lambda simple [def-env pattern body]
      ↓(ccons 'simple def-env pattern body)))

(define REFLECT
   (lambda simple [def-env pattern body]
      ↓(ccons 'reflect def-env pattern body)))
```

```
(define MACRO
   (lambda simple [def-env pattern body]
      ((lambda simple [expander]
          (lambda reflect [args env cont]
             (normalize (expander . args) env cont)))
       (simple def-env pattern body))))

(define REFLECT!
   (lambda simple [def-env pattern body]
      (let [[fun (simple def-env pattern body)]]
         (lambda reflect [args env cont]
            (normalize args env
               (lambda simple [args!]
                  (fun args! env cont)))))))

(define Y-OPERATOR
   (lambda simple [fun]
      (let [[temp (lambda simple ? ?)]]
          (block (replace ↑temp ↑(fun temp)) temp))))

(define Y*-OPERATOR
   (lambda simple funs
      (let [[temps (map (lambda simple [fun] (lambda simple ? ?)) funs)]]
         (map (lambda simple [temp fun]
                 (block (replace ↑temp ↑(fun . temps)) temp))
              temps
              funs))))

(define REFLECTIFY
   (lambda simple [fun]
      (reflect (environment-designator ↑fun) (pattern ↑fun) (body ↑fun))))

(define DEFINE
   (lambda macro [label form]
      `(block (set ,label (y-operator (lambda simple [,label] ,form)))
              ,↑label)))

(define SET
   (lambda reflect [[var binding] env cont]
      (normalise binding env
         (lambda simple [binding!]
            (block (rebind var binding! env)
                   (cont ''OK))))))

(define SETREF
   (lambda reflect! [[var! binding!] env cont]
      (block (rebind var binding! env)
             (cont ''OK))))

(define REBIND
   (lambda simple [var binding env]
      (cond [(empty env) (replace ↑env ↑[[var binding]])]
            [(= var (1st (1st env))) (rplacn 2 ↑(1st env) ↑binding)]
            [$T (rebind var binding (rest env))])))

(define LET
   (lambda macro [list body]
      `((lambda simple ,(map 1st list) ,body) . ,(map 2nd list))))

(define LETSEQ
   (lambda macro [list body]
      (if (empty list)
          body
          `(let [,(1st list)]
              (letseq ,(rest list) ,body)))))

(define LETREC
   (lambda macro [list body]
      `((lambda simple ,(map 1st list)
           (block
              (block . ,(map (lambda simple [x] `(set . ,x)) list))
              ,body))
        . ,(map (lambda simple [x] ''?) list))))
```

Control Structure Utilities

```
(define IF
   (lambda reflect [args env cont]
      ((ef (rail args)
           (lambda simple []
              (normalize (1st args) env
                 (lambda simple [premise!]
                    (normalize (ef ↓premise! (2nd args) (3rd args))
                               env
                               cont))))
           (lambda simple []
              (reduce ↑ef args env cont)))))
(define COND-HELPER
   (lambda simple [clauses env cont]
      (normalize (1st (1st clauses)) env
         (lambda simple [premise!]
            (if ↓premise!
                (normalize (2nd (1st clauses)) env cont)
                (cond-helper (rest clauses) env cont))))))
(define COND (reflectify cond-helper))
(define BLOCK-HELPER
   (lambda simple [clauses env cont]
      (if (unit clauses)
          (normalize (1st clauses) env cont)
          (normalize (1st clauses) env
             (lambda simple ?
                (block-helper (rest clauses) env cont))))))
(define BLOCK (reflectify block-helper))
(define DO
   (lambda macro args
      (let [[loop-name (acons)]
            [variables (map 1st (1st args))]
            [init (map 2nd (1st args))]
            [next (map 3rd (1st args))]
            [quitters (2nd args)]
            [body (if (double args) '$T (3rd args))]]
           `(letrec
               [[,loop-name
                    (lambda simple ,variables
                       (cond
                        . ,(append quitters
                                   `[[$T (block ,body
                                                (,loop-name . , next))]])))]]
               (,loop-name . , init)))))
(define SELECT
   (lambda macro args
      (letseq
           [[dummy (acons)]
            [select-helper
               (lambda simple [[choice action]]
                  (cond [(rail choice)
                         `[(member ,dummy ,choice) ,action]]
                        [(not (boolean choice))
                         `[(= ,dummy ,choice) ,action]]
                        [$T `[,choice ,action]]))]]
           `(let [[,dummy ,(1st args)]]
               (cond . ,(map select-helper (rest args)))))))
```

```
(define SELECTQ
   (lambda macro args
      (letseq
           [[dummy (acons)]
            [selectq-helper
               (lambda simple [[choice action]]
                  (cond [(atom choice)
                         `[(= ,dummy ,↑choice) ,action]]
                        [(rail choice)
                         `[(member ,dummy ,↑choice) ,action]]
                        [$T `[,choice ,action]]))]]
           (let [[,dummy ,(1st args)]]
                (cond . ,(map selectq-helper (rest args)))))))

(define CATCH
   (lambda reflect [[exp] env cont]
      (cont (normalize exp env id))))

(define THROW
   (lambda reflect! [[exp!] env cont] exp!))

(define DELAY
   (lambda macro [exp]
      `(lambda simple [] ,exp)))

(define FORCE
   (lambda simple [delayed-exp]
      (delayed-exp)))
```

Vector Utilities

```
(define 1ST (lambda simple [vector] (nth 1 vector)))
(define 2ND (lambda simple [vector] (nth 2 vector)))
(define 3RD (lambda simple [vector] (nth 3 vector)))
(define 4TH (lambda simple [vector] (nth 4 vector)))
(define 5TH (lambda simple [vector] (nth 5 vector)))
(define 6TH (lambda simple [vector] (nth 6 vector)))

(define REST (lambda simple [vector] (tail 1 vector)))

(define FOOT
   (lambda simple [vector]
      (tail (length vector) vector)))

(define UNIT
   (lambda simple [vector]
      (and (not (empty vector)) (empty (rest vector)))))

(define DOUBLE
   (lambda simple [vector]
      (and (not (empty vector)) (unit (rest vector)))))

(define MEMBER
   (lambda simple [element vector]
      (cond [(empty vector) $F]
            [(= element (1st vector)) $T]
            [$T (member element (rest vector))])))
```

```
(define ISOMORPHIC
   (lambda simple [e1 e2]
      (cond [(not (= (type e1) (type e2))) $F]
            [(= e1 e2) $T]
            [(rail e1)
             (or (and (empty e1) (empty e2))
                 (and (not (empty e1))
                      (not (empty e2))
                      (isomorphic (1st e1) (1st e2))
                      (isomorphic (rest e1) (rest e2))))]
            [(pair e1)
             (and (isomorphic (car e1) (car e2))
                  (isomorphic (cdr e1) (cdr e2)))]
            [(closure e1)
             (and (isomorphic (procedure-type e1)
                              (procedure-type e2))
                  (isomorphic (pattern e1) (pattern e2))
                  (isomorphic (body e1) (body e2))
                  (isomorphic (environment-designator e1)
                              (environment-designator e2)))]
            [(handle e1) (isomorphic ↓e1 ↓e2)]
            [$T $F])))

(define INDEX
   (lambda simple [element vector]
      (letrec
         [[index-helper
             (lambda simple [vector-tail position]
                (cond [(empty vector-tail) 0]
                      [(= (1st vector-tail) element) position]
                      [$T (index-helper (rest vector-tail) (1+ position))]))]]
         (index-helper vector 1))))

(define VECTOR-CONSTRUCTOR
   (lambda simple [template]
      (if (external template) scons rcons)))

(define XCONS
   (lambda simple args
      (pcons (1st args) (rcons . (rest args)))))

(define MAP
   (lambda simple args
      (cond [(empty (2nd args)) ((vector-constructor (2nd args)))]
            [(double args)
             (prep ((1st args) (1st (2nd args)))
                   (map (1st args) (rest (2nd args))))]
            [$T (prep ((1st args) . (map 1st (rest args)))
                      (map . (prep (1st args) (map rest (rest args)))))])))

(define COPY-VECTOR
   (lambda simple [vector]
      (if (empty vector)
          ((vector-constructor vector))
          (prep (1st vector) (copy-vector (rest vector))))))

(define CONCATENATE
   (lambda simple [rail1 rail2]
      (replace (foot rail1) rail2)))

(define APPEND
   (lambda simple [vector1 vector2]
      (if (empty vector1)
          vector2
          (prep (1st vector1)
                (append (rest vector1) vector2)))))

(define APPEND*
   (lambda simple args
      (if (unit args)
          (1st args)
          (append (1st args) (append* . (rest args))))))
```

```
(define REVERSE
   (letrec
      [[rev (lambda simple [v1 v2]
               (if (empty v1)
                   v2
                   (rev (rest v1) (prep (1st v1) v2))))]]
      (lambda simple [vector]
         (rev vector ((vector-constructor vector))))))

(define PUSH
   (lambda simple [element stack]
      (replace ↑stack
               ↑(prep element
                      (if (empty stack)
                          (scons)
                          (prep (1st stack) (rest stack)))))))

(define POP
   (lambda simple [stack]
      (let [[top (1st stack)]]
           (block
              (replace ↑stack ↑(rest stack))
              top))))
```

Arithmetic Utilities

```
(define 1+ (lambda simple [n] (+ n 1)))
(define 1- (lambda simple [n] (- n 1)))

(define **
   (lambda simple [m n]
      (do [[i 0 (1+ i)]
           [a 1 (* a m)]]
          [[(= i n) a]])))

(define REMAINDER
   (lambda simple [x y]
      (- x (* (/ x y) y))))

(define ABS
   (lambda simple [n]
      (if (< n 0) (- n) n)))

(define MAX
   (lambda simple numbers
      (letrec
         [[max2
             (lambda simple [x y] (if (> x y) x y))]
          [max-helper
             (lambda simple [unseen-numbers maximum]
                (if (empty unseen-numbers)
                    maximum
                    (max-helper (rest unseen-numbers)
                                (max2 maximum (1st unseen-numbers)))))]]
         (max-helper (rest numbers) (1st numbers)))))

(define MIN
   (lambda simple numbers
      (letrec
         [[min2
             (lambda simple [x y] (if (< x y) x y))]
          [min-helper
             (lambda simple [unseen-numbers minimum]
                (if (empty unseen-numbers)
                    minimum
                    (min-helper (rest unseen-numbers)
                                (min2 minimum (1st unseen-numbers)))))]]
         (min-helper (rest numbers) (1st numbers)))))

(define ODD  (lambda simple args (not (zero (remainder n 2)))))
(define EVEN (lambda simple args (zero (remainder n 2))))
```

```
(define NEGATIVE     (lambda simple [n] (<  n 0)))
(define NON-NEGATIVE (lambda simple [n] (>= n 0)))
(define POSITIVE     (lambda simple [n] (>  n 0)))
(define ZERO         (lambda simple [n] (=  n 0)))
```

General Utilities

```
(define ATOM        (lambda simple [x] (= (type x) 'atom)))
(define RAIL        (lambda simple [x] (= (type x) 'rail)))
(define PAIR        (lambda simple [x] (= (type x) 'pair)))
(define NUMERAL     (lambda simple [x] (= (type x) 'numeral)))
(define HANDLE      (lambda simple [x] (= (type x) 'handle)))
(define BOOLEAN     (lambda simple [x] (= (type x) 'boolean)))
(define CHARAT      (lambda simple [x] (= (type x) 'charat)))
(define CLOSURE     (lambda simple [x] (= (type x) 'closure)))
(define STREAMER    (lambda simple [x] (= (type x) 'streamer)))

(define NUMBER      (lambda simple [x] (= (type x) 'number)))
(define SEQUENCE    (lambda simple [x] (= (type x) 'sequence)))
(define TRUTH-VALUE (lambda simple [x] (= (type x) 'truth-value)))
(define CHARACTER   (lambda simple [x] (= (type x) 'character)))
(define FUNCTION    (lambda simple [x] (= (type x) 'function)))
(define STREAM      (lambda simple [x] (= (type x) 'stream)))

(define VECTOR
   (lambda simple [x] (member (type x) ['rail 'sequence])))

(define INTERNAL
   (lambda simple [x]
      (member (type x)
              ['atom 'rail 'pair 'numeral 'handle 'boolean 'charat
               'closure 'streamer])))

(define EXTERNAL
   (lambda simple [x]
      (member (type x) ['number 'sequence 'truth-value 'character
                        'function 'stream])))

(define CHARACTER-STRING
   (lambda simple [s]
      (cond [(or (not (sequence s)) (empty s)) $F]
            [(and (unit s) (character (1st s))) $T]
            [$T (and (character (1st s))
                     (character-string (rest s)))])))

(define ENVIRONMENT
   (lambda simple [closure]
      ↓(environment-designator closure)))

(define REFERENT
   (lambda reflect! [[exp! env!] env cont]
      (normalize ↓exp! ↓env! cont)))

(define MACRO-EXPANDER
   (lambda simple [macro-closure]
      ↓(binding 'expander (environment ↑macro-closure))))

(define ID  (lambda simple [x] x))
(define ID* (lambda simple x x))

(define QUOTE (lambda reflect [[a] e c] (c ↑a)))

(define RPLACT
   (lambda simple [n rail new-tail]
      (replace (tail n rail) new-tail)))

(define RPLACN
   (lambda simple [n rail new-element]
      (replace (tail (- n 1) rail) (prep new-element (tail n rail)))))
```

```
(define RPLACA
   (lambda simple [pair new-car]
      (replace pair (pcons new-car (cdr pair)))))
(define RPLACD
   (lambda simple [pair new-cdr]
      (replace pair (pcons (car pair) new-cdr))))
(define NOT (lambda simple [x] (if x $F $T)))
(define AND
   (lambda reflect [args env cont]
      (if (rail args)
          (and-helper args env cont)
          (normalize args env
             (lambda simple [args!]
                (and-helper args! env cont))))))
(define AND-HELPER
   (lambda simple [args env cont]
      (if (empty args)
          (cont '$T)
          (normalize (1st args) env
             (lambda simple [premise!]
                (if ↓premise!
                   (and-helper (rest args) env cont)
                   (cont '$F))))))))
(define OR
   (lambda reflect [args env cont]
      (if (rail args)
          (or-helper args env cont)
          (normalize args env
             (lambda simple [args!]
                (or-helper args! env cont))))))
(define OR-HELPER
   (lambda simple [args env cont]
      (if (empty args)
          (cont '$F)
          (normalize (1st args) env
             (lambda simple [premise!]
                (if ↓premise!
                   (cont '$T)
                   (or-helper (rest args) env cont))))))))
```

Input / Ouput

```
(define READ (lambda simple [stream] (mystery)))               ; Implemented, but not explained.
(define PRINT (lambda simple [x stream] (mystery)))            ; Implemented, but not explained.
(define INTERNALIZE (lambda simple [x] (mystery)))             ; Not yet implemented.
(define EXTERNALIZE (lambda simple [x] (mystery)))             ; Not yet implemented.
(define PRINT-STRING
   (lambda simple [string stream]
      (if (empty string)
          'OK
          (block (output (1st string) stream)
                 (print-string (rest string) stream)))))
(define NEWLINE
   (lambda simple [stream]
      (output #
stream)))
```

```
(define PROMPT&READ
   (lambda simple [level stream]
      (block (newline stream)
             (print ↑level stream)
             (print-string "> " stream)
             (read stream))))
(define PROMPT&REPLY
   (lambda simple [answer level stream]
      (block (print ↑level stream)
             (print-string "= " stream)
             (print answer stream))))
```

System

```
(define VERSION
   (lambda simple []
      "3-LISP version A00.  May 1, 1983"))
(define LOAD
   (lambda macro [filename]
      `(loadfile ,↑filename)))
(define EDIT
   (lambda macro [name]
      `(editdef ,↑name)))
```

Primitive Procedures

```
(define TYPE       (lambda simple [e] (type e)))
(define =          (lambda simple entities (= . entities)))
(define EF         (lambda simple [premise c1 c2] (ef premise c1 c2)))
(define UP         (lambda simple [e] (up e)))
(define DOWN       (lambda simple [s!] (down s!)))
(define REPLACE    (lambda simple [s1 s2] (replace s1 s2)))

(define ACONS      (lambda simple [] (acons)))

(define PCONS      (lambda simple [s1 s2] (pcons s1 s2)))
(define CAR        (lambda simple [pair] (car pair)))
(define CDR        (lambda simple [pair] (cdr pair)))

(define RCONS      (lambda simple structures (rcons . structures)))
(define SCONS      (lambda simple entities (scons . entities)))
(define PREP       (lambda simple [e vector] (prep e vector)))
(define LENGTH     (lambda simple [vector] (length vector)))
(define NTH        (lambda simple [n vector] (nth n vector)))
(define TAIL       (lambda simple [n vector] (tail n vector)))
(define EMPTY      (lambda simple [vector] (empty vector)))

(define CCONS
   (lambda simple [kind def-env pattern body]
      (ccons kind def-env pattern body)))
(define PROCEDURE-TYPE
   (lambda simple [closure] (procedure-type closure)))
(define ENVIRONMENT-DESIGNATOR
   (lambda simple [closure] (environment-designator closure)))
(define PATTERN    (lambda simple [closure] (pattern closure)))
(define BODY       (lambda simple [closure] (body closure)))

(define +          (lambda simple numbers (+ . numbers)))
(define -          (lambda simple numbers (- . numbers)))
(define /          (lambda simple [n1 n2] (/ n1 n2)))
```

```
(define *         (lambda simple numbers (* . numbers)))
(define <         (lambda simple numbers (< . numbers))))
(define <=        (lambda simple numbers (<= . numbers)))
(define >         (lambda simple numbers (> . numbers)))
(define >=        (lambda simple numbers (>= . numbers)))

(define INPUT     (lambda simple [stream] (input stream)))
(define OUTPUT    (lambda simple [e stream] (output e stream)))

(define LOADFILE  (lambda simple [file-name] (loadfile file-name)))

(define EDITDEF   (lambda simple [procedure-name] (editdef procedure-name)))
```

Appendix B. How to Implement 3-LISP

Since the 3-LISP reflective tower is infinite, and since the standard definition of 3-LISP is non-effective, neither the reflective processor nor the informal meta-theoretic descriptions of 3-LISP show how the language is finite. In this section, however, we show why 3-LISP is indeed finite, and present a program that implements a full virtual tower, as a constructive demonstration of how it can be effectively implemented. As it happens, we use 3-LISP as the implementing language, and for simplicity embed the structural field, global environment, etc., isomorphically (i.e., a rail is implemented directly as a rail, etc.). The resulting processor therefore bears the same relationship to 3-LISP as standard meta-circular processors bear to standard LISPs. The implementation makes no crucial use of the reflective capabilities of the embedding 3-LISP, and no crucial use of recursion; the code, therefore, could be straightforwardly translated into PASCAL, microcode, or any other language of choice. If one were to implement 3-LISP in such a language, however, one would have to implement the 3-LISP structural field as well.

An analysis of the 3-LISP tower is given in section B.1., showing how all but a finite number of the lowest levels contain no information. A simple but complete implementation (about 120 lines of code) is then presented in section B.2. In section B.3. we show how to "compile" other procedures into the implementation (kernels, standards, etc.), including many simples and some reflectives (`LAMBDA`, `IF`, etc.), and show how to make the control flow in the implementation processor more transparent.

B.1. The Finite Nature of 3-LISP

It is first important to understand how 3-LISP treats tail-recursion. In particular, notice that if the processor normalizes a redex of the form '`(FUN . ARGS)`' in some environment E_0 with continuation C_0, the form '`FUN`' is normalized with a C-PROC! continuation that embeds a binding of the atom '`CONT`' to C_0. Assuming that the closure that results (`FUN!`, so to speak) is not reflective, '`ARGS`' is normalized with a C-ARGS! continuation that also has '`CONT`' bound to C_0. Then, assuming that '`FUN!`' was not primitive, either, the body of the closure is normalized, in an environment built by extending the environment from '`FUN!`' by matching the pattern to `ARGS!`, *and with the continuation* C_0. In other words (as Steele and Sussman point out in the SCHEME literature), the processor continuation embeds for *argument* processing, but not for *procedure calling.*

We say, because of this continuation protocol, that the 3-LISP processor runs programs ***tail-recursively***. If, in other words, there is a call to `FOO`, for example:

```
(+ 2 (FOO X Y))
```

and `FOO` has the following definition:

```
(define FOO
   (lambda simple [a b]
      (* a (1+ b))))
```

then the continuation in effect when the expression '`(FOO X Y)`' is normalized will be identical to the one in which the body of `FOO` — '`(* a (1+ b))` — is normalized.

Generalizing slightly, we say that a *position* or *context* within an expression is ***tail-recursive*** with respect to the embedding expression if, and only if, when a sub-expression in that context is normalized in the course of normalizing the embedding context, it is normalized *with the same* *continuation as that used to normalize the whole.* We have just seen that the bodies of closures are tail-recursive with respect to full procedure calls, but there are some other cases. Specifically, consider the expression

```
(IF (= 1 2) 'YES 'NO)
```

given the following definition of IF (simplified for clarity from the standard one):

```
(define IF
   (lambda reflect [[premise c1 c2] env cont]
      (normalize premise env
         (lambda simple [premise!]
            (normalize (ef ↓premise! c1 c2) env cont)))))
```

The first argument to IF ('(= 1 2)' in the example) is normalized with a (LAMBDA SIMPLE [PREMISE!] ...) continuation, but when the premise has returned a boolean of one sort or the other, the selected consequent (C1 or C2 — 'YES or 'NO in the example) is normalized with the same continuation as was the whole IF redex. The second and third argument positions to IF, therefore, are tail-recursive with respect to the embedding IF.

We adopt the presentational convention of underlining an expression (or the left parenthesis and the CAR, if the expression is another redex) if it is in a tail-recursive context with respect to the redex it occurs within. Thus we would have the following presentation for FOO:

```
(define FOO
   (lambda simple [a b]
      (* a (1+ b))))
```

and the following definition of the normal recursive FACTORIAL (since both arguments to IF are tail-recursive with respect to IF):

```
(define FACTORIAL
   (lambda simple [n]
      (if (= n 0)
          1
          (* n (factorial (1- n))))))
```

Since the embedded call to FACTORIAL is not underlined, FACTORIAL, as a whole, will generate continuation structure ("stack") proportional to the depth of the recursion. An iterative version, however, is the following:

```
(define FACTORIAL
   (lambda simple [n]
      (factorial-helper 1 n)))

(define FACTORIAL-HELPER
   (lambda simple [acc n]
      (if (= n 0)
          acc
          (factorial-helper (* acc n) (1- n)))))
```

In this case the recursive call is underlined, since it is tail-recursive with respect to its own definition, which implies (because the *processor* runs programs tail-recursively) that no continuation structure is generated by recursive calls to FACTORIAL-HELPER, or, to put it another way, FACTORIAL-HELPER is ***iterative.***

It can be determined by simple inspection of the definitions that all of the consequent clauses of CONDS are tail-recursive with respect to the COND, as are the clauses of SELECT and SELECTQ, as well as other common constructs.

Given this analysis of tail-recursion, we then look at the processor code itself (not, this time, at what it does with the continuations for the program it is *running*, but at its *own* code, with respect to the continuations it will require *in the processor that is running it*). Specifically, we can immediately underline the tail-recursive positions in the magnificent seven. We have distinguished the continuations from the main bodies of the three named primary processor procedures by using italics and bold-face. For example, C-PROC! (lines 16–25) is shown in italics; the call to IF (line 17) is the top-level call in its body, and the calls to the de-reflected version of PROC! and to NORMALIZE are underlined, since they are tail-recursive *with respect to the C-PROC! continuation as a whole* (not with respect to REDUCE). The other three continuations are treated similarly.

```
1  (define READ-NORMALIZE-PRINT
2    (lambda simple [level env stream]
3      (normalize (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 [exp env cont]
9      (cond [(normal exp) (cont exp)]
10           [(atom exp) (cont (binding exp env))]
11           [(rail exp) (normalize-rail exp env cont)]
12           [(pair exp) (reduce (car exp) (cdr exp) env cont)])))
13 (define REDUCE
14   (lambda simple [proc args env cont]
15     (normalize proc env
16       (lambda simple [proc!]                               ; Continuation C-PROC!
17         (if (reflective proc!)
18             (↓(de-reflect proc!) args env cont)
19             (normalize args env
20               (lambda simple [args!]                       ; Continuation C-ARGS!
21                 (if (primitive proc!)
22                     (cont ↑(↓proc! . ↓args!))
23                     (normalize (body proc!)
24                                (bind (pattern proc!) args! (environment proc!))
25                                cont))))))))))
26 (define NORMALIZE-RAIL
27   (lambda simple [rail env cont]
28     (if (empty rail)
29         (cont (rcons))
30         (normalize (1st rail) env
31           (lambda simple [first!]                          ; Continuation C-FIRST!
32             (normalize-rail (rest rail) env
33               (lambda simple [rest!]                       ; Continuation C-REST!
34                 (cont (prep first! rest!))))))))))
```

The finiteness of 3-LISP now follows directly (by inspection) from this annotated code, as manifested by the following simple control and data flow argument. First, we carry out the argument ignoring the existence of line 18:

1. Note first that the four standard continuations plus C-REPLY will always be bound to the formal parameter CONT, and furthermore that that parameter will never have any other binding. This is true a) because each continuation is bound to the formal parameter CONT in the procedure to which it is first passed (C-REPLY, C-PROC!, C-ARGS!, and C-FIRST! are each third arguments to NORMALIZE, and C-REST! is a third argument to NORMALIZE-RAIL); b) because in the three places where CONT is in turn passed as an argument to a processor procedure (lines 11, 12, and 25) it is passed to a

procedure that binds it to CONT; and c) because those eight calls (lines 3, 11, 12, 15, 19, 23, 30, and 32) are the only places in the processor that the three named procedures are called.

2. Each of the nine calls to a named processor procedure (again, lines 3, 6, 11, 12, 15, 19, 23, 30, and 32) is in a tail-recursive position with respect to the procedure or continuation in which it occurs (i.e., all nine are underlined).
3. Each of the five calls to CONT (on lines 9, 10, 22, 29, and 34) are also in tail recursive positions with respect to the procedures or continuations in which they occur (they too are underlined).
4. From the previous three facts, it follows that all eight of the mutually recursive processor procedures (the magnificent seven plus READ-NORMALIZE-PRINT) always call each other tail-recursively. Therefore, it follows that the processor that is *running* this processor will build up no continuation structure by running the processor. (Actually, this is not strictly true; rather, *at each call to a ppp procedure* the continuation will be the same, but between them — as for example within a call to NORMAL — it will build up temporarily.)
5. Since (by hypothesis) all levels of the tower were initialized by the level above's reading in an expression of the form '(READ-NORMALIZE-PRINT *LEVEL* GLOBAL PRIMARY-STREAM), it follows that the continuation being passed around at each reflective level is an unchanging instance of a C-REPLY continuation (again, more accurately, this is a constant base, on top of which small excursions are constantly constructed and then discarded). All of these C-REPLYs are isomorphic except that each embeds its own binding for the variable LEVEL.
6. Since the call to ↓(DE-REFLECT PROC!) is in a tail-recursive position (underlined), the continuation that *it* will be called with by the processor running it — i.e., the continuation that will be passed to REDUCE up one level with PROC designating ↓(DE-REFLECT PROC!) — will always be a C-REPLY continuation.

If the processor contained no reflective procedures, that would be all there is to the proof. However, the processor does (crucially) contain five reflectives: AND, COND, IF, LAMBDA, and LET (it would be possible to reduce this number from five to one, but not to zero — i.e., it can be proved that the processor must contain at least one reflective closure). In order to complete the proof, therefore, we have to examine the definitions of these five procedures, and show that the de-reflected versions that are called by the processor that is running the processor share the crucial properties we just demonstrated for the basic seven procedures. LAMBDA is straightforward: its definition is:

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

It is manifest that REDUCE is called tail-recursively, and that CONT is passed to REDUCE's CONT; therefore processing (↓(DE-REFLECT ↑LAMBDA) ARGS ENV CONT) will preserve the iterative nature of the processor. Similarly COND; ↓(DE-REFLECT ↑COND) is simply COND-HELPER:

```
(define COND (reflectify cond-helper))

(define COND-HELPER
   (lambda simple [clauses env cont]
      (normalize (1st (1st clauses)) env
         (lambda simple [premise!]                     ; Continuation C-COND
            (if ↓premise!
               (normalize (2nd (1st clauses)) env cont)
               (cond-helper (rest clauses) env cont))))))
```

COND-HELPER is itself tail-recursive, calls NORMALIZE tail-recursively, passes a more complex continuation that calls NORMALIZE tail-recursively, and passes CONT only as an argument to CONT in NORMALIZE; it too, therefore, keeps the processor iterative.

AND is called in the processor only with rail arguments, so the second clause in the definition of AND is never invoked in running the processor, although it is well-formed in any case:

```
(define AND
   (lambda reflect [args env cont]
      (if (rail args)
          (and-helper args env cont)
          (normalize args env
             (lambda simple [args!]
                (and-helper args! env cont))))))
```

AND-HELPER, which is called tail-recursively with CONT passed through, calls NORMALIZE tail-recursively, with a continuation that preserves both protocols for continuations and tail-recursion.

```
(define AND-HELPER
   (lambda simple [args env cont]
      (if (empty args)
          (cont '$T)
          (normalize (1st args) env
             (lambda simple [premise!]                    ; Continuation C-AND
                (if ↓premise!
                    (and-helper (rest args) env cont)
                    (cont '$F)))))))
```

IF is very slightly more difficult to analyse, although its behavior is straightforward. The invocation of EF will construct two closures, built with LAMBDA, one of which will be selected and returned as the result of the call to EF. In constructing those closures no ppp's are called, so the processor does not embed. The *result* of the EF is the procedure that is called tail-recursively with respect to the call to ↓(DE-REFLECT ↑IF), which enables us to annotate the definition of IF as follows:

```
(define IF
   (lambda reflect [args env cont]
      ((ef (rail args)
           (lambda simple []
              (normalize (1st args) env
                 (lambda simple [premise!]                ; Continuation C-IF
                    (normalize (ef ↓premise! (2nd args) (3rd args))
                               env
                               cont))))
           (lambda simple []
              (reduce ↑ef args env cont))))))
```

Again, all calls to IF in the processor are with rail arguments, so that the middle clause is always selected. Again, the call to NORMALIZE is appropriately tail-recursive, as is the call within the provided continuation, and the continuation of the level below is passed through intact.

Finally we have LET. The definition is as follows:

```
(define LET
   (lambda macro [list body]
      `((lambda simple ,(map 1st list) ,body) . ,(map 2nd list))))
```

It is clear that no processor procedures are called at all in constructing the form to be handed back to the processor for normalization; and the form that is constructed contains only a LAMBDA, which we have already treated.

B.2. 3×3: A Direct Embedding of 3-LISP in 3-LISP

This section contains a complete implementation of 3-LISP in 3-LISP. (It has been run successfully by the current implementation, albeit *very slowly*. Furthermore, the actual INTERLISP implementation was derived from this code.)

The differences between the following implementation processor and the reflective processor are relatively minor:

1. `:NORMALIZE` is the implementation of `NORMALIZE`; `:REDUCE` implements `REDUCE`, etc.
2. All calls between the implementations of primary processor procedures are done indirectly by `CALL`ing the real version. For example, the line `(normalize-rail exp env cont)` in `NORMALIZE` becomes `(call normalize-rail exp env cont)` in `:NORMALIZE`. A quick glance at the implementation processor will reveal no explicit calls to any procedures with a name beginning in ':'.
3. The closures for the standard continuations are explicitly constructed with `MAKE-CONTINUATION`. This ensures that legitimate standard continuation closures are built (we would not want to give the object program access to an implementation level closure; `REDUCE` and `:REDUCE` are similar but not identical).
4. The four classes of standard continuations, C-PROC!, C-ARGS!, C-FIRST!, and C-REST!, are implemented by top-level procedures `:C-PROC!`, `:C-ARGS!`, `:C-FIRST!`, and `:C-REST!`, respectively. The procedure `IMPORT` is used to access non-local variables (e.g., C-PROC! uses `ARGS` so `:C-PROC!` must import `ARGS`).
5. `:C-ARGS!`, the implementation of C-ARGS!, contains additional code that shifts the implementation processor down whenever possible (i.e., whenever one of the closures for which an implementation procedures exists is about to be expanded). (The corresponding logic for shifting up a level whenever necessary is buried in `CALL`.)
6. The parameter pattern for a primary processor procedure should also be used by the implementation in order to ensure that pattern match failures happen to the implementation if, and only if, they would happen to the reflective processor.

Italics are used in the following code to indicate those fragments that differ from the corresponding code in the reflective processor.

:NORMALIZE, :REDUCE, and :NORMALIZE-RAIL

```
(define :NORMALIZE
   (lambda simple [exp env cont]
      (cond [(normal exp) (call cont exp)]
            [(atom exp) (call cont (binding exp env))]
            [(rail exp) (call normalize-rail exp env cont)]
            [(pair exp) (call reduce (car exp) (cdr exp) env cont)])))

(define :REDUCE
   (lambda simple [proc args env cont]
      (call normalize proc env
         (make-continuation @sample-c-proc!))))

(define :C-PROC!
   (lambda simple [proc!]
      (import [args env cont]
         (if (reflective proc!)
             (call ↓(de-reflect proc!) args env cont)
             (call normalize args env
                (make-continuation @sample-c-args!))))))
```

```
(define :C-ARGS!
   (lambda simple [args!]
      (import [proc! cont]
         (cond [(primitive proc!) (call cont ↑(↓proc! . ↓args!))]
               [(processor-procedure proc!)
                (block (shift-down cont)
                       (register ↓proc! ↓args!)
                       ((implementation-of proc!) . ↓args!))]
               [$T (expand-closure proc! args! cont)]))))

(define EXPAND-CLOSURE
   (lambda simple [proc! args! cont]
      (call normalize (body proc!)
                      (bind (pattern proc!) args! (environment proc!))
                      cont)))

(define :NORMALIZE-RAIL
   (lambda simple [rail env cont]
      (if (empty rail)
          (call cont (rcons))
          (call normalize (1st rail) env
             (make-continuation @sample-c-first!)))))

(define :C-FIRST!
   (lambda simple [first!]
      (import [rail env]
         (call normalize-rail (rest rail) env
            (make-continuation @sample-c-rest!)))))

(define :C-REST!
   (lambda simple [rest!]
      (import [first! cont]
         (call cont (prep first! rest!)))))
```

CALL

We can't call object-level continuations with (cont ...), since if they were reflective, that would cause the *implementation processor* to reflect, rather than enabling us to reflect the tower it is running. Similarly we can't call any of the seven primary processor procedures directly, like NORMALIZE and C-PROC!, since we need to use our own private versions of them (:NORMALIZE, :C-PROC!, etc.). Also, we can't call simple user procedures directly if they are *not* primary processor procedures, since we won't have implementation level code for them; they require that we shift up and expand their bodies explicitly.

CALL-SIMPLE, which is only used by (the expansions of) CALL, checks to see if the procedure to be called *at the current level* is a primary processor procedure. Since primary processor procedures have an implementation equivalent that can be run *at the current level*, there is no need to change levels. However, we must REGISTER this call so that we can "chicken out" later. In all other cases, lacking code to run *at the current level*, the implementation processor shifts up one level and expands the closure — i.e., runs the implementation of NORMALIZE *at the next higher level*. In effect, CALL implements both "compiled-to-compiled" and "compiled-to-interpreted" calls, where the primary processor procedures are the only "compiled" routines in the system.

By assumption, all of the primary processor procedure implementations (the ':' routines) are correct implementations of their counterparts in the reflective processor provided that no "funny business" is involved. In particular, the implementations are not designed to handle reflective continuations (if the continuation would be called; no harm is done if a reflective continuation is simply passed along to some other procedure, or embedded in a continuation). When an implementation procedure is on the verge of calling a reflective continuation, CALL will detect this fact, shift up, and expand the closure for the primary processor procedure that was making the CALL using the information recorded by REGISTER in the global variables @LAST-PROCESSOR-PROCEDURE, and @LAST-

PROCESSOR-ARGS (we refer to this process as *chickening out*).

We also have to chicken out when we encounter one of the primitive procedures being used as a continuation. One reason is that the primitives return an answer — that would cause the continuation-passing implementation processor to cease its processing. Another reason is that there might be a reflective continuation *two* levels up that should prevent the primitive from being called (see note at end of section).

```
(define CALL
   (lambda macro exp
      `(let [[fun ,(1st exp)]]
         (if (or (reflective ↑fun) (primitive ↑fun))
             (expand-closure @last-processor-procedure        ; chicken-out!
                             @last-processor-args
                             (shift-up))
             (call-simple fun ,(rest exp))))))

(define CALL-SIMPLE
   (lambda simple [fun args]
      (if (processor-procedure ↑fun)
          (block (register fun args)
                 ((implementation-of ↑fun) . args))
          (expand-closure ↑fun ↑args (shift-up)))))
```

REGISTER

Every time we enter the implementation version of a primary processor procedure we use REGISTER to record in global variables (registers) the details of the event. This information is used in three distinct ways: 1) by MAKE-CONTINUATION in constructing continuations, 2) by IMPORT as the source of non-local variable bindings, and 3) by CALL in "chickening out."

```
(define REGISTER
   (lambda simple [fun args]
      (block (set @last-processor-procedure ↑fun)
             (set @last-processor-args ↑args))))
```

MAKE-CONTINUATION

It is important that the continuation closures built by the implementation processor be indistinguishable from the ones that the reflective processor would build. In particular, all C-PROC! (say) closures share the same pattern and body structures. They also have an environment designator (rail) whose initial bindings cells are made from fresh lengths of rail, but whose fourth tail is the environment designator found in the closure for REDUCE. Also, all C-ARGS! continuation closures contain an environment designator whose first tail is the environment designator found in some (unique) C-PROC! closure. MAKE-CONTINUATION is passed a template closure, from which the appropriate pattern and body structures are extracted, and uses the globally-recorded current primary processor procedure and the arguments passed to it.

```
(define MAKE-CONTINUATION
   (lambda simple [template]
      (simple ↑(bind (pattern @last-processor-procedure)
                     @last-processor-args
                     (environment @last-processor-procedure))
              (pattern template)
              (body template))))
```

IMPORT

The standard continuations use some variables defined in an enclosing non-global scope. For example, a C-PROC! continuation uses ARGS, ENV, and CONT, which are local to REDUCE; a C-ARGS! continuation uses CONT and PROC!, which are local to REDUCE and the enclosing C-PROC!, respectively. Thus the implementations of the standard continuations need to get hold of these bindings. This is achieved by having IMPORT extract the bindings from the environment designator of the closure for the current primary processor procedure (@last-processor-procedure).

```
(define IMPORT
   (lambda macro [vars body]
      `(let ,(map (lambda simple [var]
                     `[,var ↓(binding ,↑var (environment @last-processor-procedure))])
                  vars)
            ,body)))
```

For example, the code:

```
;;;   (define :C-REST!
;;;      (lambda simple [rest!]
;;;         (import [first! cont]
;;;            (call cont (prep first! rest!)))))
```

is equivalent by this macro-expansion to the following:

```
;;;   (define :C-REST!
;;;      (lambda simple [rest!]
;;;         (let [[first! ↓(binding 'first! (environment @last-processor-procedure))]
;;;               [cont ↓(binding 'cont (environment @last-processor-procedure))]]
;;;            (call cont (prep first! rest!)))))
```

PROCESSOR-PROCEDURE

PROCESSOR-PROCEDURE is used to recognize closures that correspond to some primary processor procedure. For these procedures, IMPLEMENTATION-OF retrieves the corresponding implementation procedure that can be *called* instead of *expanding* their closure. TABLE-OF-EQUIVALENTS serves as the basis of this mapping; but a simple equality test is inadequate since each standard continuation is actually a whole *family* of closures. The procedure MATCH-CLOSURE is used to determine if a particular closure is sufficiently similar to a canonical member of its family to ensure that the implementation procedure would be a correctly implementation. "Sufficiently similar" amounts to having identical patterns and bodies, and sufficiently similar environment designators, as determined by MATCH-ENV. For MATCH-ENV to succeed, both rails must be the same length, share a tail that includes the global rail as a proper tail, and have plausible binding cells for the same atoms and in the same order.

Note that in a serious implementation it would be ludicrous to do all of this pattern matching: instead the implementation should "stamp" the processor procedure closures in a way that is invisible to 3-LISP proper, but visible to its internal version of PROCESSOR-PROCEDURE (and the stamp would be invalidated if a user ever obtained access to such a closure and smashed it). Recognition (PROCESSOR-PROCEDURE) and mapping onto implementation equivalent (IMPLEMENTATION-OF) could then be unit-time operations (but with a price: the criteria for class membership would be restricted, so that some closures isomorphic to standard processor procedures would not be recognized, even though they deserve to be).

```
(define PROCESSOR-PROCEDURE
   (lambda simple [proc]
      (do [[table table-of-equivalents (rest table)]]
          [[(empty table) $F]
           [(match-closure proc (1st (1st table))) $T]])))
```

```
(define IMPLEMENTATION-OF
   (lambda simple [proc]
      (do [[table table-of-equivalents (rest table)]]
          [[(match-closure proc (1st (1st table))) (2nd (1st table))]])))
(set TABLE-OF-EQUIVALENTS
     [[↑normalize :normalize]    [↑normalize-rail :normalize-rail]
      [↑reduce :reduce]          [@sample-c-proc! :c-proc!]
      [@sample-c-args! :c-args!] [@sample-c-first! :c-first!]
      [@sample-c-rest! :c-rest!]])
(define MATCH-CLOSURE
   (lambda simple [candidate master]
      (or (= candidate master)
          (and (= (body candidate) (body master))
               (= (pattern candidate) (pattern master))
               (match-env (environment-designator candidate)
                          (environment-designator master))))))
(define MATCH-ENV
   (lambda simple [candidate master]
      (cond [(= master ↑global) $F]
            [(= candidate master) $T]
            [$T (and (not (empty candidate))
                     (rail (1st candidate))
                     (double (1st candidate))
                     (= (1st (1st candidate)) (1st (1st master)))
                     (handle (2nd (1st candidate)))
                     (match-env (rest candidate) (rest master)))])))
```

SAMPLE CONTINUATION CLOSURES

Samples of each of the four kinds of standard continuation closures are needed (they are used with MAKE-CONTINUATION and in TABLE-OF-EQUIVALENTS). This clever way of procuring them will only work if the implementation language is a full-blown 3-LISP; in any other setting it will be necessary to apply a somewhat more tedious approach — see NEW-TOP-LEVEL-CONTINUATION for an example.

```
(define THROW-CONT (lambda reflect [[] env cont] ↑cont))

(set @SAMPLE-C-PROC! ↑(catch ((throw-cont))))
(set @SAMPLE-C-ARGS! ↑(catch (id* . (throw-cont))))
(set @SAMPLE-C-FIRST! ↑(catch ['? (throw-cont)]))
(set @SAMPLE-C-REST! (binding 'cont (environment @sample-c-first)))
```

SHIFT-UP *and* **SHIFT-DOWN**

SHIFT-UP pretends that we are now playing reflective processor at one level higher than we were just a moment ago, and adjusts the continuation stack, @LEVEL-STACK, so that it accurately reflects our new stance. Similarly, SHIFT-DOWN pretends that we are going to play reflective processor at one level lower than we were a moment ago, and saves the continuation for our former level on the continuation stack. The continuation stack should contain a continuation for each of the reflective levels; however, we postpone their creation until the implementation first reaches that reflective level.

```
(define SHIFT-UP
   (lambda simple []
      (if (empty @level-stack)
          (new-top-level-continuation)
          (pop @level-stack))))

(define SHIFT-DOWN (lambda simple [cont] (push cont @level-stack)))
```

GENESIS

GENESIS starts things off at level 1 with a continuation stack consisting entirely of top-level continuations. Note that the call to READ-NORMALIZE-PRINT will cause the implementation to shift up to level 2, although the embedded call to NORMALIZE within it will subsequently drop it back down again.

```
(define GENESIS
   (lambda simple []
      (block (set @level-stack (scons))
             (set @next-level 1)
             (call read-normalize-print 1 global primary-stream))))
```

NEW-TOP-LEVEL-CONTINUATION

The tower (hanging garden) we implement is allegedly initialized in the following way. First, "God" normalizes the form:

```
(read-normalize-print ∞ global primary-stream)
```

and then types in the following set of incantations (the form read in on each line generates the "prompt&read" for the next):

```
∞> (read-normalize-print ∞-1 global primary-stream)
...
3> (read-normalize-print 2 global primary-stream)
2> (read-normalize-print 1 global primary-stream)
```

This means that the activity at level 1 is driven by the tail-recursive (underlined) call to NORMALIZE inside READ-NORMALIZE-PRINT:

```
;;;   (define READ-NORMALIZE-PRINT
;;;      (lambda simple [level env stream]
;;;         (normalize (prompt&read level stream) env
;;;            (lambda simple [result]                     ; Continuation C-REPLY
;;;               (block (prompt&reply result level stream)
;;;                      (read-normalize-print level env stream))))))
```

Top-level continuations, then, are simply closures created by the normalization of the LAMBDA expression within READ-NORMALIZE-PRINT (italicized in the foregoing).

The only use of the global variable @NEXT-LEVEL is to set up the correct binding for LEVEL inside each successive new top level continuation, in order to simulate the infinite number of incantations. The strictly linear "hierarchy" of control levels is a partial myth, foisted on the user by this initialization protocol.

```
(define NEW-TOP-LEVEL-CONTINUATION
   (letseq [[rnp-environment (environment ↑read-normalize-print)]
            [rnp-pattern (pattern ↑read-normalize-print)]
            [rnp-body (body ↑read-normalize-print)]
            [c-reply-pattern (2nd (cdr (3rd (cdr rnp-body))))]     ; i.e., '[result]'
            [c-reply-body (3rd (cdr (3rd (cdr rnp-body))))]]       ; i.e., '(block ... stream))'
      (lambda simple []
         (block (set @next-level (1+ @next-level))
                (simple ↑(bind rnp-pattern
                               ↑[@next-level global primary-stream]
                               rnp-environment)
                        c-reply-pattern
                        c-reply-body)))))
```

NOTES ON 3×3

★ This version is presented using DEFINEs, but, in fact, if you were to run this you would have to establish all of these procedures definitions in a giant LABELS, since otherwise these definitions will be visible in the global environment, which would be incorrect. It is crucial, however, that the environment we hand out (through READ-NORMALIZE-PRINT) be the real global environment, so that when user code reflects, it gets access to the genuine article.

★ The implementation assumes that the object level program will be prevented from smashing those parts of the standard 3-LISP system upon which it depends. For example, the implementation would die if the object program smashed SET since the implementation uses SET on a regular basis (in REGISTER), and even though it is conceivable that the underlying implementation need not have protected SET since it isn't in the kernel. Conversely, everything that is protected in the underlying implementation is, like it or not, protected in the new tower.

★ CALL is defined as a macro because it is critical that the argument expression not be processed when the procedures being CALLed is either reflective or primitive and the argument processing potentially involves a side-effect or an error.

★ In the reflective processor, the check for reflective closures is performed in C-PROC!, not C-ARGS!. As a consequence, any closure that makes it to C-ARGS! as the binding of PROC! will be expanded regardless of its procedure type. In other words, in regular 3-LISP the expression (FOO (REPLACE ↑FOO ↑(REFLECTIFY FOO))) will treat FOO as if it were a simple closure (which it was at the time C-PROC! had a look at it). It is for this reason that MATCH-CLOSURE ignores procedure type.

★ The implementations are correct only relative to the standard reflective processor — :NORMALIZE does not engender the behavior of just any old program walking over the body of the closure for NORMALIZE.

★ The viability of the technique of chickening out depends on the fact that primary processor procedures do nothing irrevocable prior to calling their continuation. When this is not the case, it is necessary to do a more *vertical* shift-up; this involves putting together authentic-looking environment and continuation structures describing the *current* state of the computation one level up and shifting up into :C-PROC!. Chickening out causes a shift up into the tail-end of :C-ARGS! at an earlier instant.

★ To make sure that the bindings are in the right order (e.g., PROC, then ARGS, then ENV, then CONT), MAKE-CONTINUATION uses the same kernel procedure (BIND) as the reflective processor.

★ The call to BIND in MAKE-CONTINUATION will not fail provided that the primary processor procedures and their implementations have similar patterns.

SOME NASTY TEST CASES

The 3-LISP reflective processer provides a fine-grained description of how 3-LISP programs are processed. An implementation of 3-LISP can be considered *correct* only if careful attention is paid to the many subtleties entailed by this account. Here are some nasty test cases that illuminate some of the finer points that are easily missed.

1. Replacing a simple closure with a reflective one. The test for reflectiveness is done prior to normalizing the arguments. Hence, changing a simple closure into a reflective one during argument normalization will not have an immediate effect.

```
1> (set foo (lambda simple [x] (+ x 1)))
1= 'foo
1> (set fee (lambda reflect [x] (- x 1)))
1= 'fee
1> (foo 100)
1= 101
1> (foo (block (replace ↑foo ↑fee) 100))
1= 99
1> (foo 100)
{ERROR: Pattern match failure}
```

2. Using reflective procedures as continuations. The effect of using a reflective procedure as a continuation is bizarre but predictable!

```
1> (normalize '1 global id)
1= '1
1> (normalize '1 global quote)
1= 'exp                              ; From line 9 of reflective processor
1> (normalize '+ global id)
1= '{simple + closure}
1> (normalize '+ global quote)
1= '(binding exp env)                ; From line 10
1> (normalize '(+ 1 2) global id)
1= '3
1> (normalize '(+ 1 2) global quote)
1= '↑(↓proc! . ↓args!)               ; From line 22
1> (normalize-rail '[] global id)
1= '[]
1> (normalize-rail '[] global quote)
1= '(rcons)                          ; From line 29
1> (normalize-rail '[1] global id)
1= '[]
1> (normalize-rail '[1] global quote)
1= '(prep first! rest!)              ; From line 34
```

3. Smashing a continuation. An implementation may not trust the procedure type of a continuation — it can be changed on the fly.

```
1> (let [[dummy-id (lambda simple [x] x)]]
      (normalize `(id (replace ,↑↑dummy-id ↑quote)) global dummy-id))
1= '(binding exp env)
1> (let [[dummy-id (lambda simple [x] x)]]
      (normalize `(id (replace ,↑↑dummy-id ↑up)) global dummy-id))
1= '''OK
```

4. Tampering with the environment of a continuation closure. The environment designator within a standard continuation closure can be changed, making it non-standard. In the following, a new binding for NORMALIZE is stuffed into a C-ARGS! closure; this binding will be used when it comes to expanding the closure.

```
1> (define CHANGE-CONT
      (lambda reflect [[exp] env cont]
         (block
            (push ['normalize ↑(lambda simple [a e c] (c ↑a))]
                  (environment ↑cont))
            (normalize exp env cont))))
1= 'CHANGE-CONT
1> ((lambda simple x (print 'hello primary-stream)) . (change-cont (+ 2 2)))
1= '(print 'hello primary-stream)
```

5. Sharing of environment tails between C-PROC! and C-ARGS! continuations. A C-ARGS! closure contains an environment designator whose first tail is the environment designator captured by the corresponding C-PROC! closure.

```
1> (define SAVE-CONT
     (lambda reflect [[var exp] env cont]
       (block (rebind var ↑↑cont env)
              (normalize exp env cont))))
1= 'SAVE-CONT
1> ((save-cont x1 -) . (save-cont x2 [1]))          ; i.e. (- 1)
1= -1
1> x1
1= '{simple C-PROC! closure}
1> x2
1= '{simple C-ARGS! closure}
1> (= (environment-designator x1) (rest (environment-designator x2)))
1= $T
```

6. Fresh top-level continuations. Each time through READ-NORMALIZE-PRINT a new C-REPLY continuation closure is created.

```
1> (define SAVE-CONT
     (lambda reflect [[var exp] env cont]
       (block (rebind var ↑↑cont env)
              (normalize exp env cont))))
1= 'SAVE-CONT
1> (save-cont x1 x1)
1= '{simple C-REPLY closure}
1> (save-cont x2 x2)
1= '{simple C-REPLY closure}
1> (= x1 x2)
1= $F
1> (= (pattern x1) (pattern x2))
1= $T
1> (= (body x1) (body x2))
1= $T
1> (= (environment-designator x1) (environment-designator x2))
1= $F
1> (= (environment x1) (environment x2))
1= $T
```

7. Using a primitive as a continuation with a reflective continuation over it. Care must be taken in such cases because the primitive may never get invoked.

```
1> (normalize '(normalize '10 global output) global quote)
1= '↑(↓proc! . ↓args!)              ; From line 22 of the reflective processor
```

8. Rebinding a kernel procedure in the global environment. This is almost always fatal.

```
1> (set normalize 10)
```
[Thud.]

9. Smashing a kernel procedure, its body, or its pattern. This too is usually fatal.

```
1> (set x (body ↑atom))
1= 'OK
1> x
1= '(= (type x) 'atom)
1> (rplaca x 'rcons)
```
[Thud.]

10. Clobbering the global environment. The global environment rail must always be in normal form; otherwise, (environment proc!) on line 24 would error on all standard procedures.

```
1> (replace (foot ↑global) '[ha!])
```
[Thud.]

11. Circular rails can cause NORMALIZE to hang — even normal-form ones.

```
1> (set x (rcons '1))
1= '[1]
1> (block (replace (foot x) x) 'done)          ; x = '[1 1 1 ... ]
1= 'DONE
1> (block (normalize x global id) 'done)
```

[Stuck in NORMAL-RAIL *chasing a TAIL.]*

B.3. Some Simple 3×3 Optimizations

COMPILED SIMPLES

The most glaring inefficiency in the code given in section B.2 is that, except for the seven primary processor procedures, every 3-LISP procedure is treated by explicitly expanding the closure. 3×3 can be extended so as to "compile" some standard procedures — i.e., treat them in a manner similar to primitives, the only difference being that weird continuations will not cause feather dusters to be donned but, instead, will force the closure to be expanded. Some rules apply: most notably, no compiled procedure may call a non-compiled one (e.g., MAP and Y-OPERATOR are out) on this simple strategy.

We have to add a test to :C-ARGS! to check for procedures other than primitives for which we have "compilations" (and check to make sure that running the compiled version is "safe"), and provide a recognition mechanism. Since our implementation language is a full 3-LISP, we automatically have compilations for all simple kernels:

```
(define :C-ARGS!
   (lambda simple [args!]
      (import [proc! cont]
         (cond [(or (primitive proc!)
                    (and (compiled proc!)
                         (not (reflective ↑cont))
                         (not (primitive ↑cont))))
                 (call cont ↑(↓proc! . ↓args!))]
                [(processor-procedure proc!)
                 (block (shift-down cont)
                        (register ↓proc! ↓args!)
                        ((implementation-of proc!) . ↓args!))]
                [$T (expand-closure proc! args! cont)]))))

(define COMPILED
   (lambda simple [proc]
      (member proc compiled-procedures)))

(set COMPILED-PROCEDURES
     (map up
        [** 1+ 1- 1st 2nd 3rd 4th 5th 6th abs append append* atom bind binding
         boolean character character-string charat closure concatenate copy-vector
         de-reflect double environment even external foot function handle id id*
         index internal isomorphic macro macro-expander max member min negative
         newline non-negative normal normal-rail not number numeral odd pair pop
         positive primitive print prompt&read prompt&reply push rail read rebind
         reflect reflect! reflectify reflective remainder rest reverse rplaca
         rplacd rplacn rplact sequence simple stream streamer truth-value unit
         vector vector-constructor xcons zero]))
```

COMPILED KERNEL REFLECTIVES

To handle kernel reflectives (such as IF) one needs in general a) to define implementation procedures for the main body of the reflective procedure and for each of the continuations it constructs (of which IF has one), b) to construct a sample closure for those continuations and for the de-reflected version of the main procedure, and c) to add an appropriate entry to the TABLE-OF-EQUIVALENTS. It is essential that the compiled kernel reflective not fall off of its parentheses (for this reason, the above technique would not apply to THROW).

LAMBDA is easy; one only need add (again we use italics to indicate those parts of this implementation version that differ from the user-visible version):

```
(define :LAMBDA
   (lambda simple [[kind pattern body] env cont]
      (call reduce kind ↑[↑env pattern body] env cont)))
```

and add one more entry to the table of equivalents:

```
(set TABLE-OF-EQUIVALENTS [[↑normalize :normalize]
                           [↑normalize-rail :normalize-rail]
                           [↑reduce :reduce]
                           [@sample-c-proc! :c-proc!]
                           [@sample-c-args! :c-args!]
                           [@sample-c-first! :c-first!]
                           [@sample-c-rest! :c-rest!]
                           [(de-reflect ↑lambda) :lambda]])
```

To deal with IF, we would add (with the same use of italics):

```
(define :IF
   (lambda simple [args env cont]
      (if (rail args)
          (call normalize (1st args) env
             (make-continuation @sample-c-if))
          (call reduce ↑ef args env cont))))

(define :C-IF
   (lambda simple [premise!]
      (import [args env cont]
         (call normalize (if ↓premise! (2nd args) (3rd args)) env cont))))

(set @SAMPLE-C-IF ↑(catch (if (throw-cont) ? ?)))
```

and append the following two entries to the table of equivalents:

```
[(de-reflect ↑if) :if]
[@sample-c-if :c-if]
```

As a final example, consider compiling READ-NORMALIZE-PRINT. First, define the standard implementation version:

```
(define :READ-NORMALIZE-PRINT
   (lambda simple [level env stream]
      (call normalize (prompt&read level stream) env
         (make-continuation @sample-c-reply))))

(define :C-REPLY
   (lambda simple [result]
      (import [level env stream]
         (block (prompt&reply result level stream)
                (call read-normalize-print level env stream)))))

(set @SAMPLE-C-REPLY
   (block (set @next-level 1)
          ↑(new-top-level-continuation)))
```

Then add to the table of equivalents the entries:

```
[↑read-normalize-print :read-normalize-print]
[@sample-c-reply :c-reply]
```

AND, OR, COND, BLOCK, and so on are all similar.

To illustrate the compilation of macros, we will show how to compile DEFINE, assuming the following definition:

```
;;;   (define DEFINE
;;;      (lambda macro [label body]
;;;         `(block (set ,label (y-operator (lambda simple [,label] ,body)))
;;;                ,↑label)))
```

Note that this definition does not make accessible, to any instance of it, a rail that is shared by all definitions (i.e., it sets up no "own" variables). If it did, we would have to extract a handle to that very rail; as it is, we can construct a fresh version:

```
(define :DEFINE
   (lambda simple [[label body] env cont]
      (call normalize `(block (set ,label (y-operator (lambda simple [,label] ,body)))
                              ,↑label)
                      env
                      cont)))
```

And the standard addition to the table of equivalents:

```
[(de-reflect ↑define) :define]
```

Note that this compiles only the first stage of the macro expansion.

CONTROL FLOW

The code presented in section B.2. is inefficient in a particular way: CALL, which can be called with any kind of procedure (simple or reflective, primary processor or user) is sometimes used in a place where the argument is known to be a specific one of the three named processor procedures (NORMALIZE, REDUCE, or NORMALIZE-RAIL). In such a circumstance the code, as written, will go through a whole set of unnecessary checks to make sure that it isn't primitive or reflective, look up the implementation version, and then register the state and call that implementation version. At the point of call, however, we know perfectly well what that implementation procedure will be (specifically, for NORMALIZE it is :NORMALIZE, for REDUCE it is :REDUCE, and for NORMALIZE-RAIL it is :NORMALIZE-RAIL). It is possible, therefore, to simplify the CALL sequence considerably in these specific cases. A simple way to do so is to define three procedures (CALL-NORMALIZE, CALL-REDUCE, and CALL-NORMALIZE-RAIL) which merely do the necessary state registration and call the implementing versions directly:

```
(define CALL-NORMALIZE
   (lambda simple args
      (block (register normalize args) (:normalize . args))))

(define CALL-REDUCE
   (lambda simple args
      (block (register reduce args) (:reduce . args))))

(define CALL-NORMALIZE-RAIL
   (lambda simple args
      (block (register normalize-rail args) (:normalize-rail . args))))
```

Then, each place in the code there is an expression of the form '(CALL NORMALIZE ...)', it can be replaced with '(CALL-NORMALIZE ...)'. Rather than rewrite the whole B.2. processor, we give just those procedures that change under this revision, with the altered fragments of the code underlined. Additionally, we use CALL-SIMPLE in place of CALL in C-PROC! for reflectives, since DE-REFLECT will always return a non-primitive simple. Also, we call EXPAND-CLOSURE on READ-NORMALIZE-PRINT in GENESIS, since we know that READ-NORMALIZE-PRINT is not a processor procedure or compiled (although it can be compiled, in which case it should read (CALL-READ-NORMALIZE-PRINT ...)):

```
(define :NORMALIZE
   (lambda simple [exp env cont]
      (cond [(normal exp) (call cont exp)]
            [(atom exp) (call cont (binding exp env))]
            [(rail exp) (call-normalize-rail exp env cont)]
            [(pair exp) (call-reduce (car exp) (cdr exp) env cont)])))

(define :REDUCE
   (lambda simple [proc args env cont]
      (call-normalize proc env
         (make-continuation @sample-c-proc!))))

(define :C-PROC!
   (lambda simple [proc!]
      (import [args env cont]
         (if (reflective proc!)
             (call-simple ↓(de-reflect proc!) [args env cont])
             (call-normalize args env
                (make-continuation @sample-c-args!))))))

(define :NORMALIZE-RAIL
   (lambda simple [rail env cont]
      (if (empty rail)
          (call cont (rcons))
          (call-normalize (1st rail) env
             (make-continuation @sample-c-first!)))))

(define :C-FIRST!
   (lambda simple [first!]
      (import [rail env]
         (call-normalize-rail (rest rail) env
            (make-continuation @sample-c-rest!)))))

(define EXPAND-CLOSURE
   (lambda simple [proc! args! cont]
      (call-normalize (body proc!)
                      (bind (pattern proc!) args! (environment proc!))
                      cont)))

(define GENESIS
   (lambda simple []
      (block (set @level-stack (scons))
             (set @next-level 1)
             (expand-closure ↑read-normalize-print
                             ↑[1 global primary-stream]
                             (shift-up)))))
```

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 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.

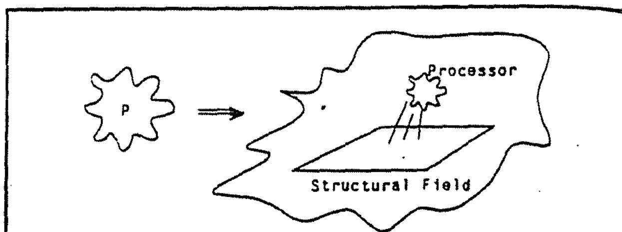

Figure 1: A Serial Model of Computation

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

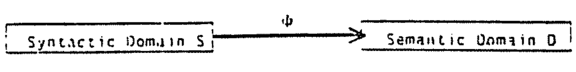

Figure 2: A Simple Semantic Intepretation Function

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 Θ, 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 n_1 and n_2 are intended to be linguistic or communicative entities, as described above; s_1 and s_2 are internal structures over which the internal processing is defined. The relationship Θ, 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 n_1 *notates* s_1). For example, in first-order logic n_1 and n_2 would be expressions, perhaps written with letters and spaces and '∃' signs; s_1 and s_2, to the extent they can even be said to exist, would be something like abstract derivation tree

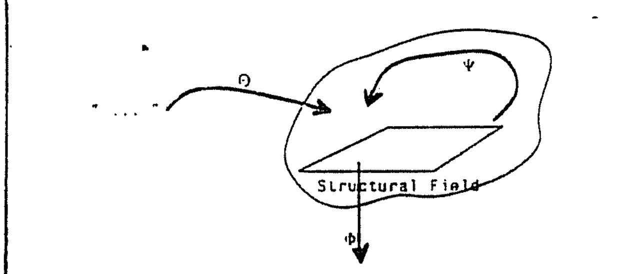

Figure 3: Semantic Relationships in a Computational Process

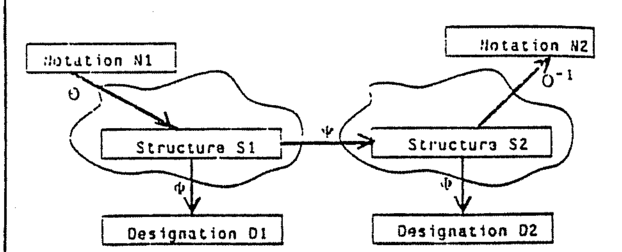

Figure 4: A Framework for Computational Semantics

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 $\Phi \circ \Theta$). 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 $\Phi(\Theta(\text{"T.S.ELIOT"})) = \text{T.S.ELIOT}$. 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 Θ 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 Θ, Ψ, and Φ have different relative importances in different linguistic disciplines, and different relationships among them have been given different names. For example, Θ is usually ignored in logic, and there is little tendency to view the study of Ψ, called proof theory, as semantical, although it is always related to semantics, as in proving soundness and completeness (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 Θ is usually specifiable compositionally and independently of Φ or Ψ is essentially a statement of the autonomy thesis for language. Similarly, when Θ 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 Θ, 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 $\Phi(\Theta(\mathbf{b})) = \mathbf{c}$. 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 $\Psi(\Theta(\mathbf{b})) = \mathbf{c}$, 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 Θ and Θ^{-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 $\Theta^{-1}\circ\Psi\circ\Theta$ (i.e., if n_1 is an expression you type at Lisp, then n_2 is $\Theta^{-1}(\Psi(\Theta(n_1)))$). 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 (Θ, Φ, and Ψ) figure crucially in an account of Lisp is not a formal one. It makes an empirical claim on the minds of programmers, and cannot be settled by pointing to any current theories or implementations. 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

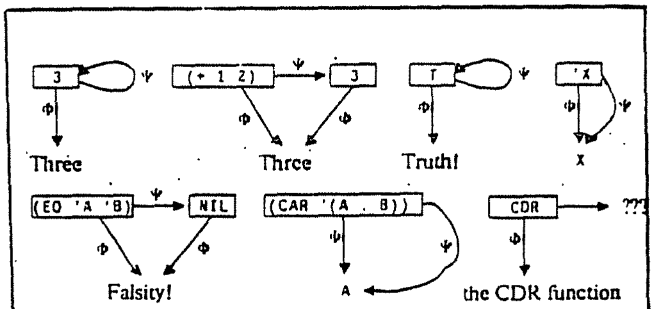

Figure 5: LISP Evaluation vs. Designation: Some Examples

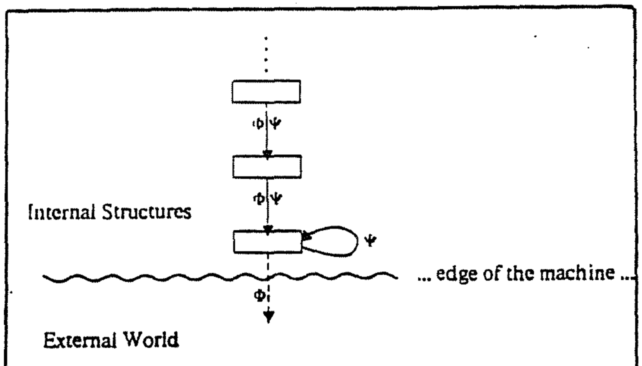

Figure 6: LISP's "De-reference If You Can" Evaluation Protocol

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.

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

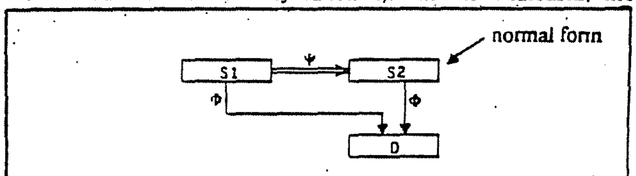

Figure 7: A Normalisation Protocol

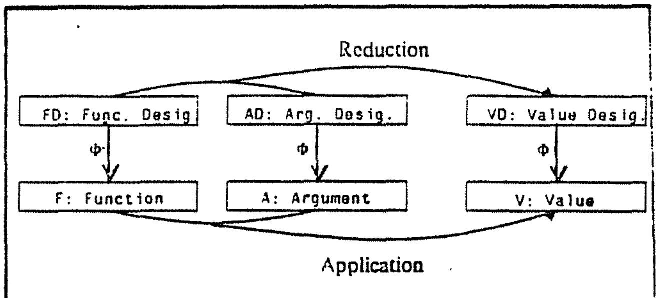

Figure 8: Application vs. Reduction

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 [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,

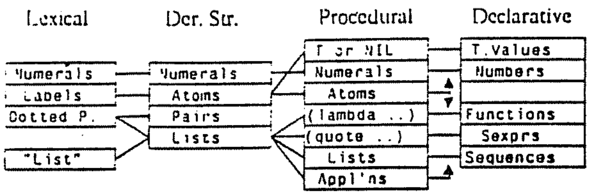

Figure 9: The Category Structure of LISP 1.5

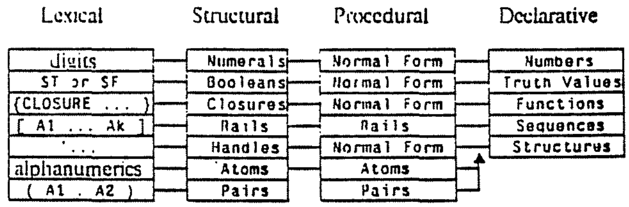

Figure 10: The Category Structure of 2-LISP and 3-LISP

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 '[A₁ A₂ ... Aₖ]') 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 '(A₁ . A₂)') designate the value of the function designated by the CAR applied to the arguments designated by the CDR. By taking the notational form '(A₁ A₂ ... Aₖ)' to abbreviate '(A₁ . [A₂ A₃ ... Aₖ])' instead of '(A₁ . (A₂ . (... (Aₖ . 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 (ATOM, 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

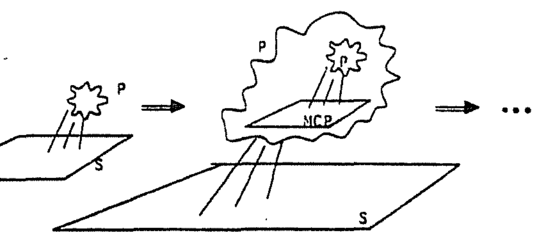

Figure 12: Meta-Circular Processors

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

More generally, the basic idea is that $\Phi(\texttt{NORMALISE}) = \Psi$, to be contrasted with $\Phi(\downarrow)$, which is approximately Φ, except that because $\downarrow$ is a partial function we have $\Phi(\downarrow \circ \texttt{NORMALISE}) = \Phi$. 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 $P \land Q$ is explained simply by conjoining the explanation of P and Q — specifically, in such a formula as: '$P \land Q$' 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 struc) (cont strc)]
          [(atom strc) (cont (binding strc env))]
          [(rail strc) (normalise-rail struc 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.

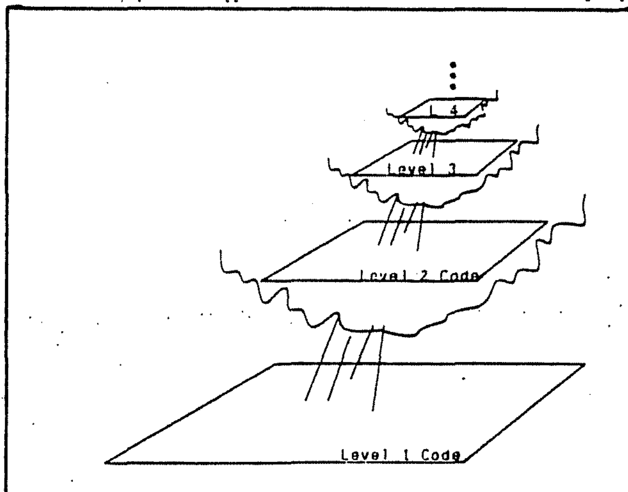

Figure 15: The 3-LISP Reflective Tower

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) $\Rightarrow$ 3; (+ 11 (three)) $\Rightarrow$ 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)) $\Rightarrow$ $T, whereas (BOUND X) $\Rightarrow$ $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 [[premise 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) $\Rightarrow$ '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.

```
(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 struc) (cont strc)]
          [(atom strc) (cont (binding strc env))]
          [(rail strc) (normalise-rail struc 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.

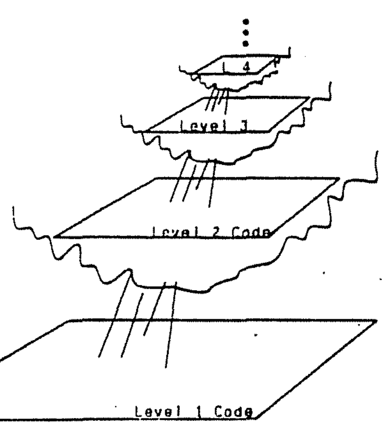

Figure 15: The 3-LISP Reflective Tower

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 [[premise 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 $\Phi \circ O$, 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., "Prolegomena to a Theory of Mechanized Formal Reasoning", *Artificial Intelligence* 13:1,2 (1980) pp. 133-170.

Standard Procedure Index