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
```

2011-05-20
IINPUT, OUTPUT and streams are not available. Instead, plaese use READ and PRINT in the following way to read from and write to STD IO: (read), (print '123), (print 'abc), (print (read)), etc.

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
```

2011-05-20
This one takes about 12 minutes on a Mac Pro with 3Ghz Xeon processor.

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
```

2011-05-20
LABELS here should be LETREC instead.

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

```
1> (TYPE '1)
1= 'NUMERAL
1> (TYPE '$T)
1= 'BOOLEAN
1> (TYPE '[1 2 3])
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:

2011-05-20 A few things about "NORMALIZE": (1) it spells with an S -- this is a Canadian version, but feel free to put "(set normalize normalise)" in the file init.3lisp; (2) NORMALISE takes 3 arguments, the last 2 being an environment and a continuation -- you can use "global" and "id" for these in these examples if you are short of arguments.

2011-05-20
(3) In these 2 examples, environment is given as [[...] [...] ...]. In 3LispR, env is explicitly constructed and expanded in the following ways:
(bind '[a b] '[1 2] (econs))
(bind 'a '1 (bind 'b '2
  (bind 'add ^+ (econs))))


```
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:

```
 7 ..... (define NORMALIZE
 8 ..... (lambda simple [exp env cont]
 9 ..... (cond [(normal exp) (cont exp)]
10 ..... [(atom exp) (cont (binding exp env))]
11 ..... [(rail exp) (normalise-rail exp env cont)]
12 ..... [(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 components 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):

```
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)))))))))
```

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:

```
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-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:

(LET [[V_1 E_1][V_2 E_2] ... [V_n E_n]] *BODY*)

expands to

((LAMBDA SIMPLE [V_1 V_2 ... V_n] *BODY*) E_1 E_2 ... 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.)