An Implementation of 2-Lisp

by

Charles D. Halpern

Computer Science Department
Indiana University
Bloomington, IN 47405

TECHNICAL REPORT NO. 160

An Implementation of 2-Lisp

June, 1984

by

Charles D. Halpern

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

An Implementation of 2-Lisp

Charles D. Halpern

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, Indiana 47405

Abstract

2-Lisp is a dialect of Lisp designed by Brian Smith. The goal of this project is to provide a convenient exposure to 2-Lisp. To this end, this paper begins with a short introduction to 2-Lisp. Also, an annotated implementation of 2-Lisp, written in Franz Lisp, is provided.

1. INTRODUCTION

2-Lisp is a dialect of Lisp designed by Brian Smith [Smith 82]. This paper includes an introduction to 2-Lisp and an implementation of 2-Lisp written in Franz Lisp. The introduction to 2-Lisp includes an analogy between the simplification of fractions and the normalization of 2-Lisp structures. Simple interactions with the processor are shown. After the 2-Lisp processor's standard mode of input and output are used, the nonstandard modes are shown. These nonstandard modes reveal how the 2-Lisp processor uses Franz Lisp objects to represent the 2-Lisp structures it reads and prints. Finally, the complete code of the implementation is presented, with annotations.

2. AN INTRODUCTION TO 2-LISP

Previous introductions to 2-Lisp have emphasized the contrasts with other languages. This introduction does not take that approach. Similarities, not contrasts, are shown.

The 2-Lisp processor operates interactively. The user types something to the machine, the machine types something in return, and the cycle repeats itself. This mode of interaction is familiar to those acquainted with Lisp's read-eval-print loop. The processing done by the 2-Lisp machine between reading and printing is called *normalization*. The normalization of 2-Lisp expressions can be compared to the *simplification* of fractions.

This material is based on work supported by the National Science Foundation under grant numbers MCS 83-04567 and MCS 83-03325.

1. Simplification. For the purposes of this paper, these are three different *fractions*:

$$\frac{6}{3} \qquad \frac{2}{1} \qquad \frac{4}{10}$$

And these are two different *rationals*:

$$6 \div 3 \qquad 2 \qquad 2 \div 1 \qquad 0.4 \qquad 4 \div 10$$

The equals sign is used to show identity.

$$\frac{6}{3} \neq \frac{2}{1} \neq 2 \div 1 = 2$$

The fractions are thought of as concrete structures, the rationals as Platonic ideals. Each fraction designates exactly one rational. More than one fraction designates the same rational. A vertical single arrow shows this relationship between sign and signified.

$$\begin{tikzcd}
\frac{6}{3} \arrow[d] & & & & \frac{2}{1} \arrow[d] & & & & \frac{4}{10} \arrow[d] \\
6 \div 3 & = & 2 & = & 2 \div 1 & \neq & 0.4 & = & 4 \div 10
\end{tikzcd}$$

Not every downward arrow that could have been drawn has been drawn. It's important to realize that since $6 \div 3$, 2, and $2 \div 1$ are one and the same, this diagram is also correct:

$$\begin{tikzcd}
\frac{6}{3} \arrow[d] & & & & \frac{2}{1} \arrow[d] & & & & \frac{4}{10} \arrow[d] \\
2 \div 1 & = & 6 \div 3 & = & 2 & \neq & 0.4 & = & 4 \div 10
\end{tikzcd}$$

$\frac{6}{3}$ and $\frac{2}{1}$, though different fractions, designate the same rational. They are called *co-designators*. A fraction may be in *simple-form*. $\frac{1}{2}$ is in simple-form; $\frac{6}{3}$ and $\frac{4}{10}$ are not. *Simplification* is the process that takes a fraction and returns a fraction in simple-form that designates the same rational as the original. Simplification takes fractions to their simple-form co-designators. A horizontal double arrow shows simplification.

$$\begin{tikzcd}
\frac{6}{3} \arrow[d] \arrow[rrrr, Rightarrow] & & & & \frac{2}{1} \arrow[d] & & & & \frac{4}{10} \arrow[d] \\
2 \div 1 & = & 6 \div 3 & = & 2 & \neq & 0.4 & = & 4 \div 10
\end{tikzcd}$$

2. The Fraction Processor. An interactive fraction processor would consist of a read-simplify-print loop. It would begin by prompting the user.

Fraction ?

The user would respond with a fraction to be simplified.

Fraction ? $\frac{6}{3}$

The processor would then read the fraction, print a simple-form co-designator of the fraction, and then cycle back to the prompt.

Fraction ? $\frac{6}{3}$
==> $\frac{2}{1}$
Fraction ?

If the fraction that was entered was already in simple-form, the processor's job would be especially easy.

Fraction ? $\frac{6}{3}$
==> $\frac{2}{1}$
Fraction ? $\frac{2}{1}$
==> $\frac{2}{1}$
Fraction ?

Rationals are never typed by the user or printed by the processor. Only fractions are actually manipulated.

3. Normalization. In 2-Lisp, 2-Lisp structures take the place of fractions. *Numerals*, *booleans*, *closures*, *rails*, *handles*, *atoms*, *pairs*, and *maps* are the eight types of 2-Lisp structures. The place of the rationals is taken by *numbers*, *truth values*, *functions*, *sequences*, *environments* and the eight types of 2-Lisp structures. Numerals designate numbers, booleans designate truth values, closures designate functions, rails designate sequences, maps designate environments, and handles designate 2-Lisp structures.

The numeral 57 designates the number 57.

$$\begin{tikzcd}\mathbf{57} \arrow[d] \\ \mathbf{57}\end{tikzcd}$$

Numerals are already in normal-form, so they can normalize to themselves. There is only one numeral for each number, so they must.

$$\begin{tikzcd} \texttt{57} \arrow[r, Rightarrow] \arrow[d] & \texttt{57} \arrow[d] \\ 57 \arrow[r, phantom, "="] & 57 \end{tikzcd}$$

Booleans, like numerals, are already in normal-form.

$$\begin{tikzcd} \texttt{\$T} \arrow[r, Rightarrow] \arrow[d] & \texttt{\$T} \arrow[d] \\ \textit{Truth} \arrow[r, phantom, "="] & \textit{Truth} \end{tikzcd}$$

Closures designate functions [Smith 84]. The atoms that are the names of the primitive operations normalize to closures and designate functions.

$$\begin{tikzcd} \texttt{+} \arrow[r, Rightarrow] \arrow[d] & \texttt{\{expr prim [n1 n2] Franz\}} \arrow[d] \\ \textit{the plus function} \arrow[r, phantom, "="] & \textit{the plus function} \end{tikzcd}$$

A rail designates the sequence of the designations of the elements of the rail. Thus, the rail `[$T $F]` designates the 2-element sequence ⟨ *Truth, Falsity* ⟩.

$$\begin{tikzcd} \texttt{[\$T \$F]} \arrow[d] & {} \arrow[d, phantom, "\text{because}"] & \texttt{\$T} \arrow[d] & {} \arrow[d, phantom, "\text{and}"] & \texttt{\$F} \arrow[d] \\ \langle \textit{Truth}, \textit{Falsity} \rangle & {} & \textit{Truth} & {} & \textit{Falsity} \end{tikzcd}$$

Handles are 2-Lisp structures that designate other 2-Lisp structures. For example, the handle `'$F` designates the structure `$F`. There is exactly one handle for each structure, one structure for each handle. All handles are in normal-form.

$$\begin{tikzcd} \texttt{'\$F} \arrow[r, Rightarrow] \arrow[d] & \texttt{'\$F} \arrow[d] \\ \texttt{\$F} \arrow[r, phantom, "="] & \texttt{\$F} \end{tikzcd}$$

Atoms may designate anything. An atom designates the same thing that its binding (in the appropriate environment) designates. Atoms are never in normal-form. An atom always normalizes to its binding. If the atom `ident` is bound to the numeral 13, the following diagram is correct.

$$\begin{tikzcd}
\texttt{ident} \arrow[r, Rightarrow] \arrow[d] & \mathbf{13} \arrow[d] \\
\mathbf{13} \arrow[r, phantom, "="] & \mathbf{13}
\end{tikzcd}$$

Pairs, like atoms, may designate anything. Pairs designate the application of the designation of the left hand side to the designation of the right hand side. If a pair is to have a designation, its car must designate a function and its cdr must designate an argument. Just as 6 ÷ 3 was 2 in the simplification subsection, so the application of a function to a sequence may *be* anything in any of the thirteen categories listed in the first paragraph of this subsection.

$$\begin{tikzcd}
\texttt{(+ . [2 3])} \arrow[r, phantom, "="] \arrow[d] & \texttt{(+ 2 3)} \arrow[r, Rightarrow] & \texttt{5} \arrow[d] \\
\textit{the plus function}\text{ applied to }\langle 2,3\rangle \arrow[rr, phantom, "="] & & \mathbf{5}
\end{tikzcd}$$

A pair may designate a Platonic number, as above–or a concrete rail structure, below.

$$\begin{tikzcd}
\texttt{(rcons '(+ 2 3) ''\$T)} \arrow[r, Rightarrow] \arrow[d] & \texttt{'[(+ 2 3) '\$T]} \arrow[d] \\
\textit{the rcons function}\text{ applied to }\langle\texttt{(+ 2 3)},\texttt{'\$T}\rangle \arrow[r, phantom, "="] & \texttt{[(+ 2 3) '\$T]}
\end{tikzcd}$$

4. The 2-Lisp processor. The 2-Lisp processor has a read-normalize-print loop in place of the read-simplify-print loop of the fraction processor. It begins by prompting the user.

```
2-Lisp ?
```

The user responds with a structure to be normalized.

```
2-Lisp ?  (rcons '(+ 2 3) ''$T)
```

The processor then reads the 2-Lisp structure (a pair in this case), prints a normal-form co-designator of the structure, and cycles back to the prompt.

```
2-Lisp ?  (rcons '(+ 2 3) ''$T)
     ==>  '[(+ 2 3) '$T]
2-Lisp ?
```

If the structure that is entered is already in normal-form, the processor's job is especially easy.

```
2-Lisp ?  (+ 2 3)
     ==>  5
2-Lisp ?  5
     ==>  5
2-Lisp ?
```

Note that the 2-Lisp processor reads the *numeral* 5 and prints the *numeral* 5. The *number* 5 is neither read nor printed.

3. THE REPRESENTATION

The following examples use the override facilities to show how Franz Lisp objects are used to represent 2-Lisp structures. Overrides are entered by typing a tilde followed by a Franz expression. The first override used is the `~echo` override. This tells the 2-Lisp processor not to normalize what it reads, but rather to just print it out again.

```
2-Lisp ?  ~echo
     ==>  Override: echo
2-Lisp ?  (+ 2 3)
       =  (+ 2 3)
2-Lisp ?
```

The `~print` override causes the processor to print out the Franz Lisp representation of the 2-Lisp structure. With both the `~echo` and the `~print` overrides in effect, the 2-Lisp processor becomes a machine that shows the internal representation of each 2-Lisp expressions it reads.

```
2-Lisp ?  ~echo
     ==>  Override: echo
2-Lisp ?  ~print
       =  Override: print
2-Lisp ?  57
       =  (N . 57)
2-Lisp ?  $T
       =  (B . t)
2-Lisp ?  [$T $F]
       =  (R (B . t) R (B) R)
2-Lisp ?  (+ 2 3)
       =  (P (A . +) R (N . 2) R (N . 3) R)
2-Lisp ?
```

The details of this representation scheme appear in the Implementation section of this paper.

4. DESIGN GOALS

The primary design goal for this implementation is convenience. It is designed for the person who'd like to try the examples in Smith [82]. Two aspects of convenience are stressed in this paper.

1. Conformity. The first aspect of convenience is conformity. The user should be able to type in examples just as they appear in Smith [82]. This requires that all of 2-Lisp's special syntax be supported. The use of brackets for rails, dollar signs for booleans, single quotes for handles, etc. are all carefully mimicked in this implementation.

Motivated by the goal of convenience, one major break with Smith [82] is made. Instead of using Smith's names for most primitives, an extension of the Scheme 84 [Friedman *et al.* 84] taxonomy is used. `set!` replaces `set`, for example. A side-by-side listing of Smith's names and their replacements can be found in an appendix. Since most users of this implementation are accustomed to Scheme 84, this decision can be viewed as a decision in favor of conformity. This implementation conforms to what most fingers want to type. The Scheme 84 naming-system is slightly extended, however. For example, the exclamation mark is used in the names of *all* functions with side effects. This fits in with Smith's theme of uniformity.

2. Speed. The second aspect of convenience is speed. The user shouldn't have to wait all day to see a simple example processed. This requirement means that time/space trade-offs are uniformly decided in favor of time.

Smith [82] includes a MacLisp implementation of 3-Lisp as an appendix. That implementation represents atoms with MacLisp atoms, numerals with MacLisp numbers, and uses fewer tags then this one to represent rails. This implementation tags all representations. The numeral 57 is represented by `(N . 57)` for example. This uses more space, but it means that much can be done with simple table dispatches.

The effect of this next speed-increasing technique is part real, part psychological. Each time this implementation prompts the user, it prints its prompt and then does a garbage collection before reading. If the input is long (a function definition, say), the garbage is collected by the time the user is finished. This is parallel processing on a small scale. If the input is short–and typed immediately–the processor takes slightly longer to respond than normal (it has to finish the garbage collection first). The printing of the prompt *before* the once-a-cycle garbage collection makes the user feel like the machine is responding quickly. This trick helps make up for the extensive use of tags (garbage).

5. THE IMPLEMENTATION

The 2-Lisp processor is loaded by loading the file `/usiu/charlie/2/s/all.l`. (The 2 is for 2-Lisp, the `s` is for speed.) This file loads all the other files. It will provide an outline for the following discussion.

```
(load "/usiu/charlie/.lisprc")

(chdir "/usiu/charlie/2/s")
        (load "m/manage.l")
        (load "m/alist.l")

        (load "f/manage.l")
        (load "f/read.l")
        (load "f/write.l")
        (load "f/normal.l")
        (load "f/alist.l")
        (load "f/primitives.l")
        (load "f/initalist.l")
        (load "f/loop.l")
```

First the files containing macros are loaded, then the files containing functions.

1. m/manage.l. These macros create and manipulate representations of 2-Lisp structures, such as handles and rails. `define-syntax` is not actually used to define the macros; `defmacro` is used. Definitions using `define-syntax` are easier to read because they avoid the use of backquotes and commas. `define-syntax` is just another name for `mkmac` [Kohlbecker 84].

Each representation is tagged with a symbol (Franz atom) that corresponds to the type of the 2-Lisp structure it represents. Given a representation, `tag-pt` returns this symbol. The tag will be one of the following: `N`, `B`, `C`, `R`, `H`, `A`, `P`, `O`, `M`.

```
(define-syntax
   (tag-pt struc)
   (car struc))
```

`nume?` is a predicate that checks whether its argument is the representation of a 2-Lisp numeral. Numerals are represented by Franz Lisp fixnums, appropriately tagged. `mk-nume` yields a representation of a 2-Lisp numeral from a fixnum. `nfixnum` takes a representation of a 2-Lisp numeral, and strips off the tag–returning a fixnum.

```
(define-syntax
   (nume? struc)
   (eq 'N (car struc)))

(define-syntax
   (mk-nume fixnum)
   (cons 'N fixnum))

(define-syntax
   (nfixnum nume)
   (cdr nume))
```

Up to this point in the paper, Franz Lisp objects are referred to as representations of 2-Lisp structures. For example the Franz Lisp `(N . 7)` is said to *represent* the 2-Lisp numeral 7. Since only one way of representing 2-Lisp structures with Franz objects is used in this implementation, it is convenient to identify Franz objects with the 2-Lisp structures they represent. From this point on, `(N . 7)` *is* the numeral 7. Similarly, the Franz `(B . nil)` is the 2-Lisp boolean `$F`; and `(B . t)`, `(B . 57)`, *etc.* are *all* the 2-Lisp structure `$T`. Note that this paper also blurs the distinction between notations and internal structures.

```
(define-syntax
   (bool? struc)
   (eq 'B (car struc)))

(define-syntax
   (mk-bool tnil)
   (cons 'B tnil))

(define-syntax
   (btnil bool)
   (cdr bool))
```

The closure structure is tagged with a `C` and includes a type, environment, pattern and body. The type is a Franz symbol: `expr`, `impr` or `macro`. The environment is a Franz a-list that matches Franz symbols to 2-Lisp structures. The pattern is a 2-Lisp rail-atom tree. (A rail-atom tree has rails for internal nodes and atoms for leaves.) The body is a 2-Lisp expression.

```
(define-syntax
   (clos? struc)
   (eq 'C (car struc)))

(define-syntax
   (mk-clos type list pattern body)
   (cons 'C (cons type (cons alist (cons pattern body)))))

(define-syntax
   (ctype clos)
   (cadr clos))

(define-syntax
   (calist clos)
   (caddr clos))
```

Since primitives need only the primitive environment, the following trick can be used. The environment field of a primitive closure does not contain an a-list but rather the symbol `prim`. This indicates that the body field contains Franz code instead of the usual 2-Lisp. `cprim?` is the predicate that checks whether a closure is primitive.

```
(define-syntax
   (cprim? proc)
   (eq (calist proc) 'prim))

(define-syntax
   (cpattern clos)
   (cadddr clos))

(define-syntax
   (cbody clos)
   (cddddr clos))
```

Next come rails. If $s_1, s_2, \ldots, s_n$ are 2-Lisp structures, then `(R` s_1 `R` s_2 `R` ... `R` s_n `R)` is a 2-Lisp rail structure. Notice that the entire structure is tagged with an `R`, as is each tail.

Smith [82] also inserts `R`'s. A rail starts out with only one `R`, and more `R`'s are `replaced` in when tails are manipulated. Hence there can be any number of `R`'s–anywhere in the representation. This results in the heavy use of functions that skip over all the initial `R`'s, in order to find what they're really looking for. That representation may use less space in most situations, but this one allows quick manipulation of rail structures. This design decision was made keeping in mind the goal of this project, to attain reasonable speeds on small examples.

There are at least two things to note about the `rcons` macro. First, its name. `rcons` is mnemonic for rail constructor. But it isn't named `mk-rail`, which would be in the tradition of `mk-nume`, `mk-bool`, and `mk-clos`. This is because there is an important difference between the `mk-` functions and the `cons` functions. The `mk-` functions make 2-Lisp structures out of Franz objects. Fixnums, a-lists, *etc.* are all non-2-Lisp objects. The arguments to `cons` functions, on the other hand, are always 2-Lisp structures themselves.

Second, note that `rcons` can take any number of arguments. It does this by taking advantage of the fact that a macro takes as its single argument the entire unevaluated expression that invoked it. `rcons-h` is then applied to the list of arguments; this constructs the code that replaces the macro invocation.

```
(defun rcons-h (args)
   (if (null args)
       '(cons 'R nil)
       '(cons 'R (cons ,(first args) ,(rcons-h (rest args))))))
```

```
(define-syntax
   (rail? struc)
   (eq 'R (car struc)))

(def rcons (macro (l)
   (let ([args (cdr l)])
      (rcons-h args))))
```

`xcons` constructs redexes (from 2-Lisp structures--thus `cons`, not `mk-`). For example,

```
(xcons '(A . plus) '(N . 3) '(N . 4))
```

returns

```
(P (A . plus) R (N . 3) R (N . 4))
```

which in 2-Lisp notation is `(plus . [3 4])`, or more commonly `(plus 3 4)`.

```
(def xcons (macro (l)
    (let ([proc (cadr l)]
          [args (cddr l)])
       `(pcons ,proc ,(rcons-h args)))))
```

A primitive gets its arguments packaged as a single rail structure. The first four elements of this structure are frequently needed. Thus, in the interest of speed, `(rnth3 ···)` is just a fast equivalent of `(rnth 3 ···)`.

```
(define-syntax
   (rnth1 rail)
   (cadr rail))

(define-syntax
   (rnth2 rail)
   (cadr (cddr rail)))

(define-syntax
   (rnth3 rail)
   (cadr (cddr (cddr rail))))

(define-syntax
   (rnth4 rail)
   (cadr (cddr (cddr (cddr rail)))))
```

`rfirst`, `rrest` and `rprep` are the `car`, `cdr` and `cons` familiar to all.

```
(define-syntax
   (rfirst rail)
   (cadr rail))

(define-syntax
   (rrest rail)
   (cddr rail))

(define-syntax
   (rprep first rest)
   (cons 'R (cons first rest)))

(define-syntax
   (rempty? rail)
   (null (cdr rail)))

(define-syntax
   (runit? rail)
   (and (cdr rail) (null (cdddr rail))))
```

Handles are straightforward. `hcons` puts a handle on a structure. `hstruc` removes this handle.

```
(define-syntax
   (hand? struc)
   (eq 'H (car struc)))

(define-syntax
   (hcons struc)
   (cons 'H struc))

(define-syntax
   (hstruc hand)
   (cdr hand))
```

Notice again that it's `mk-atom`, not `acons`.

```
(define-syntax
   (atom? struc)
   (eq 'A (car struc)))

(define-syntax
   (mk-atom symbol)
   (cons 'A symbol))

(define-syntax
   (asymbol atom)
   (cdr atom))
```

Pairs illustrate the `mk-`/`cons` distinction beautifully: there's both a `mk-pair` and a `pcons`. When a Franz list is transformed into a 2-Lisp pair, `mk-pair` is used. When two 2-Lisp structures are paired, `pcons` is used.

```
(define-syntax
   (pair? struc)
   (eq 'P (car struc)))

(define-syntax
   (pcons a d)
   (cons 'P (cons a d)))

(define-syntax
   (pcar pair)
   (cadr pair))

(define-syntax
   (pcdr pair)
   (cddr pair))

(define-syntax
   (mk-pair list)
   (cons 'P list))

(define-syntax
   (plist pair)
   (cdr pair))
```

Overrides are Franz objects that pretend to be 2-Lisp structures so that the 2-Lisp processor won't choke on them. They are used to toggle readtables and other similar features.

```
(define-syntax
   (over?  struc)
   (eq 'O (car struc)))

(define-syntax
   (mk-over franz)
   (cons 'O franz))

(define-syntax
   (ofranz over)
   (cdr over))
```

Maps are environment designators. Smith [82] uses rails as environment designators; environments are identified with sequences. A case could be made for using closures as environment designators; environments would be identified with functions. Rails are rejected in this implementation because the act of clobbering a 2-Lisp environment designator should clobber the corresponding internal a-list. This would be difficult to do with rails. This problem is solved by making maps nothing more than tagged a-lists. This solution has the added convenience of allowing 2-Lisp map-manipulating primitives to run with a-list speed.

```
(define-syntax
   (map? struc)
   (eq 'M (car struc)))

(define-syntax
   (mk-map alist)
   (cons 'M alist))

(define-syntax
   (malist map)
   (cdr map))

(define-syntax
   (mcons newer-map older-map)
   (mk-map (append (malist newer-map) (malist older-map))))
```

2. m/alist.l. There are several macros for dealing with association lists. The first macro takes a 2-Lisp atom and an a-list, and returns the atom's binding in the a-list. Since the a-lists match Franz symbols not 2-Lisp atoms, `binding` first strips the tag of the 2-Lisp atom that it is given. Because symbols are used instead of atoms, `assq` can be used rather than `assoc` to find the appropriate term. (*Term* is used in this context to refer to one of the dotted pairs in the a-list.) `assq` (which uses `eq`) is faster than `assoc` (which uses `equal`). `assq` returns `nil` if no term matches.

```
(define-syntax
   (binding atom alist)
   (let* ([id (asymbol atom)]
          [lu (assq id alist)])
      (if lu
         (cdr lu)
         (error (append '(binding: unbound variable:)
                   (list (asymbol atom)))))))
```

`assq` searches from left to right, so newer bindings can be pushed on an a-list by appending them to the left of the older bindings.

```
(define-syntax
   (ext older-alist newer-alist)
   (append newer-alist older-alist))
```

The 2-Lisp processor pattern matches a closure's formal parameter and the actual argument. `bind` takes a formal parameter pattern, a pattern of bindings that are to match the formals, and an a-list to extend with the new terms. `match` takes a rail-atom tree and a rail-binding tree and produces a symbol-to-binding a-list.

```
(define-syntax
   (bind pattern binding alist)
   (ext alist (match pattern binding)))
```

`set-binding!` starts out the same as `binding`. If it finds a matching term, though, it clobbers the old binding with the new. If it doesn't find a matching term, it creates one and clobbers the a-list with this new term.

```
(define-syntax
   (set-binding! atom new-binding alist)
   (let* ([id (asymbol atom)]
          [lu (assq id alist)])
      (if lu
         (rplacd lu new-binding)
         (nconc alist (list (cons id new-binding))))))
```

hr-to-rh transforms a handle on a rail to a rail of handles. **maprfirst** is to rails what **mapcar** is to lists. **hcons-f** does the same thing that **hcons** does, except it is a function instead of a macro.

```
(define-syntax
   (hr-to-rh hand)
   (let ([s (hstruc hand)])
      (if (rail? s)
          (maprfirst 'hcons-f s)
          (error "hr-to-rh"))))
```

3. f/manage.l. This is the first file to contains functions. **type** summarizes the nine tags. The **M** tags map structures. Maps designate environments, just as numerals designate numbers and closures designate functions.

```
(defun type (struc)
   (caseq (tag-pt struc)
          [N 'numeral]
          [B 'boolean]
          [C 'closure]
          [R 'rail]
          [H 'handle]
          [A 'atom]
          [P 'pair]
          [O 'override]
          [M 'map]
          [t (error "type: unknown tag")]))
```

The accepted rules of good programming style dictate that the representation of 2-Lisp structures be hidden by `type` and the other manipulation functions. `tag-pt` is used rather than `type`, however. This is done for reasons of speed, but it also makes for nicely formatted dispatch tables. The following definition provides an example of this.

```
(defun 2-equal? (s1 s2)
   (and (eq (tag-pt s1) (tag-pt s2))
        (caseq (tag-pt s1)
               [H (2-equal? (hstruc s1) (hstruc s2))]
               [C (eq s1 s2)]
               [R (eq s1 s2)]
               [P (eq s1 s2)]
               [M (eq s1 s2)]
               [N (eq (nfixnum    s1)  (nfixnum    s2))]
               [B (eq (not (btnil s1)) (not (btnil s2)))]
               [A (eq (asymbol     s1)  (asymbol     s2))]
               [O (eq (ofranz      s1)  (ofranz      s2))]
               [t (error "2-equal?: unknown tag")])))
```

`2-equal?` uses `not` on the boolean line in order to make all non-`nil`s `eq` each other. Notice that composite structures may print the same but not be equal. Atomic structures are different. There is only one numeral for each number; there are only two booleans; two atoms that print the same are the same.

```
(defun rcons-1 (list)
   (if (null list)
       (list 'R)
       (cons 'R (cons (first list) (rcons-1 (rest list))))))

(defun rlength (rail)
   (if (rempty? rail)
       0
       (add1 (rlength (rrest rail)))))

(defun rnth (n rail)
   (cond [(> 1 n) (error "rnth: index less than 1")]
         [(rempty? rail) (error "rnth: index greater than length of rail")]
         [(onep n) (rfirst rail)]
         [t (rnth (sub1 n) (rrest rail))]))
```

`rset-n!` clobbers the *n*th element of a rail. `rset-t!` clobbers the *n*th tail of a rail.

```
(defun rset-n! (n rail new-element)
   (progn (rset-n!-h n rail new-element)
          rail))

(defun rset-n!-h (n rail new-element)
   (cond [(> 1 n) (error "rset-n!: index less than 1")]
         [(rempty? rail) (error "rset-n!: index greater than length of rail")]
         [(onep n) (rplaca (cdr rail) new-element)]
         [t (rset-n!-h (sub1 n) (rrest rail) new-element)]))

(defun rset-t! (n rail new-tail)
   (progn (rset-t!-h n rail new-tail)
          rail))

(defun rset-t!-h (n rail new-tail)
   (cond [(> 0 n) (error "rset-t!: index less than 0")]
         [(zerop n) (rplacd rail (cdr new-tail))]
         [(rempty? rail) (error "rset-t!: index greater than length of rail")]
         [t (rset-t!-h (sub1 n) (rrest rail) new-tail)]))
```

`maprfirst` is used by `hr-to-rh` in `m/alist.l` above.

```
(defun maprfirst (fun rail)
   (if (rempty? rail)
       (rcons)
       (rprep (funcall  fun (rfirst rail))
              (maprfirst fun (rrest  rail)))))

(defun rreverse (rail)
   (rreverse-h rail (rcons)))

(defun rreverse-h (rail ans)
   (if (rempty? rail)
       ans
       (rreverse-h (rrest rail) (rprep (rfirst rail) ans))))
```

`rlast` returns the last element of its argument. `rstart` returns the rail that includes everything but the last element of its argument.

```
(defun rlast (rail)
   (if (runit? rail)
       (rfirst rail)
       (rlast (rrest rail))))

(defun rstart (rail)
   (if (runit? rail)
       (rcons)
       (rprep (rfirst rail) (rstart (rrest rail)))))

(defun hcons-f (struc) (hcons struc))
```

`pset-car!` strips the tag from its argument, clobbers it with a `set-car!`, and replaces the tag. `pset-cdr!`, on the other hand, is difficult to understand.

```
(defun pset-car! (pair struc)
   (mk-pair (rplaca (plist pair) struc)))

(defun pset-cdr! (pair struc)
   (mk-pair (rplacd (plist pair) struc)))
```

4. f/read.l. The goal of this project is convenience. It is not convenient to type `(P (A . plus) R (N . 2) R (N . 3) R)` when `(plus 2 3)` is intended. Thus this file becomes an exercise in readtables. First, a new readtable is created and the old one is given a name.

```
(setq 2-readtable (makereadtable nil))

(setq Franz-readtable readtable)
```

Then the new readtable is altered to treat some characters differently. `$`, `{`, `[`, `'`, `(`, `~`, `%`, `^` , and `_` signal that what follows must be read in a special way. `}`, `]`, `.`, and `)` are to be treated as ordinary Franz symbols.

```
(let ([readtable 2-readtable])
     (setsyntax '|{| 'macro '(lambda () (read-clos)))
     (setsyntax '|}| 'macro '(lambda () '|}|))
     (setsyntax '|[| 'macro '(lambda () (read-rail)))
     (setsyntax '|]| 'macro '(lambda () '|]|))
     (setsyntax '|'| 'macro '(lambda () (read-hand)))
     (setsyntax '|(| 'macro '(lambda () (read-pair)))
     (setsyntax '|.| 'macro '(lambda () '|.|))
     (setsyntax '|)| 'macro '(lambda () '|)|))
     (setsyntax '|~| 'macro '(lambda () (read-over)))
     (setsyntax '|%| 'macro '(lambda () (read-map)))
     (setsyntax '|^| 'macro '(lambda () (read-up)))
     (setsyntax '|_| 'macro '(lambda () (read-down))))
```

When a left brace is encountered, control is passed to `read-clos`. `read-clos` knows that the 2-Lisp readtable is already in effect, so it uses `2-read*` instead of `2-read`. The left brace has been passed, so the next thing read is a 2-Lisp atom, either `expr`, `impr` or `macro`. The tag is stripped off, leaving a Franz symbol. The next thing read is a map. The tag is stripped off the map to reveal an a-list. The pattern is a rail-atom tree. The body is a 2-Lisp structure. After these four parts are read, the closing brace is skipped over. The four parts are now packaged up and tagged as a closure.

```
(defun read-clos ()
   (let* ([type (asymbol (2-read*))]
          [alist (malist (2-read*))]
          [pattern       (2-read*)]
          [body          (2-read*)]
          [rightbrace       (read)])
         (mk-clos type alist pattern body)))
```

Rails are read by accumulating all the elements in one list and feeding this list to `rcons-1`. The end of a rail can be signalled by either a right bracket or a right parenthesis. The right parenthesis is included to handle input like `(plus 2 3)`, where a rail is implied. `read-pair` takes care of the `plus` and then calls `read-rail` in this situation.

```
(defun read-rail ()
   (do ([acc nil (append acc (list in))]
        [in  (2-read*) (2-read*)])
       [(or (eq in '|]|)
            (eq in '|)|))
           (rcons-1 acc)]))
```

Notice that 'x does not expand into `(quote x)`. It becomes a 2-Lisp handle structure.

```
(defun read-hand ()
   (let ([struc (2-read*)])
        (hcons struc)))
```

When a pair is read, it may take one of two forms. Either it will be a proper dotted pair like `(plus . [2 3])` or it will be in the abbreviated dotless style, `(plus 2 3)`. Either way, the left hand side of the pair is read first. If a dot comes next, then the right hand side is read. The right parenthesis is discarded. Remember the 2-readtable treats the dot and the right parenthesis like ordinary symbols. If a right parenthesis has been read, the right hand side is the empty rail. Otherwise, the first element of the implied rail has been read, and **read-rail** is called to read the rest. **read-rail** skips over the closing parenthesis. Now the left and right sides are paired.

```
(defun read-pair ()
   (let* ([lhs  (2-read*)]
          [look (2-read*)]
          [rhs  (cond [(eq '|.| look)
                       (let* ([rhs      (2-read*)]
                              [rightpar (2-read*)]])
                          rhs)]
                      [(eq '|)| look) (rcons)]
                      [t (rprep look (read-rail))])])
      (pcons lhs rhs)))
```

Overrides are Franz objects that pretend to be 2-Lisp structures. To read an override, the readtable is switched to the Franz readtable, a Franz s-expression is read and tagged, and the 2-Lisp readtable is restored. There's one exception. If the user should find herself in the nasty situation where the 2-Lisp readtable is running but 2-Lisp isn't, a ~~ will disable the 2-Lisp readtable.

```
(defun read-over ()
   (progn (setq readtable Franz-readtable)
          (let ([in (read)])
               (if (eq '|~| in)
                   (progn (print "back to Franz readtable")
                          (terpri))
                   (progn (setq readtable 2-readtable)
                          (mk-over in))))))
```

Maps, environment designators, are an addition to 2-Lisp, and so require both justification and explanation. The justification appears in the section devoted to `m/manage.1`. The explanation is done below with an example.

This is the notation for a map structure: `%[[x 2] [y 3] [x 4]]`. This map designates an environment where `x` is bound to the numeral 2 (designates the number 2) and `y` is bound to the numeral 3 (designates the number 3). It should be easy to see how this notation can be converted into an a-list. The `%` character was chosen because it looks like the notation commonly used for substitution in environments, as in $\rho[x/id]$.

```
(defun read-map ()
   (mk-map (rail-to-alist (2-read*))))

(defun rail-to-alist (rail)
   (cond [(rempty? rail) nil]
         [t (cons (rail-to-term  (rfirst rail))
                  (rail-to-alist (rrest  rail)))]))

(defun rail-to-term (rail)
   (cons (asymbol (rfirst rail))
         (rnth2 rail)))
```

Up and down arrows (^ and _) expand into `up` and `down` just like the single quote (') expands into `quote` in Franz Lisp.

```
(defun read-up ()
   (let ([arg (2-read*)])
        (pcons (mk-atom 'up) (rcons arg))))

(defun read-down ()
   (let ([arg (2-read*)])
        (pcons (mk-atom 'down) (rcons arg))))
```

The only difference between `2-read` and `2-read*` is that `2-read*` assumes that the 2-Lisp readtable is already in place.

```
(defun 2-read ()
   (let ([readtable 2-readtable])
        (2-read*)))

(defun 2-read* ()
   (theta-fix (read)))
```

Closures, rails, handles, pairs, overrides and maps all start with reserved characters that can be caught by the readtable mechanism. Booleans start with a special character too. Unfortunately, when this special character is stripped from `$T` one is left with the tricky-to-handle `T`. Whether booleans can be caught easily or not, numerals and atoms certainly can't be. This means that the readtable mechanism cannot fully do the job of *theta*, the function that takes notations to internal structures. `theta-fix` fills in this gap.

```
(defun theta-fix (in)
   (cond [(eq in '|]|) '|]|]
         [(eq in '|.|) '|.|]
         [(eq in '|)|) '|)|]
         [(numberp in) (mk-nume in)]
         [(eq in '$T)  (mk-bool t)]
         [(eq in '$t)  (mk-bool t)]
         [(eq in '$F)  (mk-bool nil)]
         [(eq in '$f)  (mk-bool nil)]
         [(atom in)    (mk-atom in)]
         [t            in]))
```

5. f/write.l. These are the functions that print out 2-Lisp structures with all the special characters. The internal representation may be printed with `1-print`. But the interest lies in `2-print`. A dispatch table is the natural way to proceed. Divide and conquer.

```
(defun 1-print (struc)
   (print struc))

(defun 2-print (struc)
   (selectq (tag-pt struc)
      [N (print-nume struc)]
      [B (print-bool struc)]
      [C (print-clos struc)]
      [R (print-rail struc)]
      [H (print-hand struc)]
      [A (print-atom struc)]
      [P (print-pair struc)]
      [O (print-over struc)]
      [M (print-map  struc)]
      [otherwise (error "2-print: unknown tag")]))
```

Franz Lisp is perfectly able to print fixnums.

```
(defun print-nume (nume)
   (patom (nfixnum nume)))
```

Booleans are also easy to handle.

```
(defun print-bool (bool)
   (if (btnil bool)
       (patom '|$T|)
       (patom '|$F|)))
```

When a closure is printed, instead of printing out the environment, either `prim` or `%map` is printed, as appropriate. In 2-Lisp, all primitives have Franz code where their bodies belong.

```
(defun print-clos (clos)
   (progn (princ '|{|)
          (patom (ctype clos))
          (cond [(cprim? clos) (patom '| prim |)]
                [t             (patom '| %map | )])
          (2-print (cpattern clos))
          (princ '| |)
          (caseq (ctype clos)
                 [(expr impr macro) (if (cprim? clos)
                                        (patom '|Franz|)
                                        (2-print (cbody clos)))]
                 [t (2-print (cbody clos))])
          (princ '|}|)))
```

`print-rail` is straightforward. `print-rail-guts` has been given a separate existence so that it can be used elsewhere.

```
(defun print-rail (rail)
   (progn (princ '|[|)
          (print-rail-guts rail)
          (princ '|]|)))

(defun print-rail-guts (rail)
   (do ([left rail (rrest left)])
        [(rempty? left) 'done]
       (2-print (rfirst left))
       (if (not (runit? left))
           (princ '| |))))
```

To print a handle, first print a single quote mark. Then print the rest.

```
(defun print-hand (hand)
   (progn (princ '|'|)
          (2-print (hstruc hand))))
```

Printing atoms is exactly the same as printing numerals.

```
(defun print-atom (atom)
   (patom (asymbol atom)))
```

There are two ways to print a pair, provided the right hand side is a rail. It can be printed with or without a dot. This is decided by the global **dot-flag**.

```
(defun print-pair (pair)
   (if (and (rail? (pcdr pair))
            (not dot-flag))
       (print-pair-not pair)
       (print-pair-dot pair)))

(defun print-pair-dot (pair)
   (progn (princ '|(|)
            (2-print (pcar pair))
          (patom '| . |)
          (2-print (pcdr pair))
          (princ '|)|)))
```

If the pair is going to be printed without a dot, then the square brackets around the rail on the right hand side must also be dropped. This is where **print-rail-guts** is used.

```
(defun print-pair-not (pair)
   (progn (princ '|(|)
          (2-print (pcar pair))
          (princ '| |)
          (print-rail-guts (pcdr pair))
          (princ '|)|)))
```

Overrides are only expected to be used at the top level.

```
(defun print-over (over)
   (progn (patom '|Override: |)
          (print (ofranz over))))
```

Printing maps follows a pattern symmetrical with reading them. But the process is flipped across the a-list/rail axis.

```
(defun print-map (map)
   (progn (princ '|%|)
          (2-print (alist-to-rail (malist map)))))

(defun alist-to-rail (alist)
   (if (null alist)
       (rcons)
       (rprep (term-to-rail  (car alist))
              (alist-to-rail (cdr alist)))))

(defun term-to-rail (term)
   (rcons (mk-atom (car term))
          (cdr term)))
```

6. f/normal.l. This file can be considered the heart of the matter. Numerals, booleans, closures, handles and maps all normalize to themselves. So do overrides. One can think of overrides as Franz Lisp voyagers making a fantastic journey through the 2-Lisp processor. In a ship consisting of an O and a cons-cell.

```
(defun normalize (struc r)
   (caseq (tag-pt struc)
      [N struc]
      [B struc]
      [C struc]
      [R (normalize-rail struc r)]
      [H struc]
      [A (binding struc r)]
      [P (reduce (pcar struc) (pcdr struc) r)]
      [O struc]
      [M struc]
      [t (error "normalize: unknown tag")]))
```

The only structures with which the 2-Lisp processor does anything are rails, atoms, and pairs. For rails, the processor just calls itself recursively. Atoms get replaced with their bindings. Pairs get reduced, which is where the work is. Note that this implementation does not handle tail recursion properly.

```
(defun normalize-rail (rail r)
   (if (rempty? rail)
       (rcons)
       (rprep (normalize      (rfirst rail) r)
              (normalize-rail (rrest  rail) r))))
```

`reduce` needs the following three macros, which turn out to be operationally identical. The variable name `args!` contains normalized arguments; `args` contains arguments that have not been normalized.

```
(defmacro reduce-prim-expr (proc! args! alist)
   '(funcall (cbody ,proc!) ,args! ,alist))

(defmacro reduce-prim-impr (proc! args alist)
   '(funcall (cbody ,proc!) ,args ,alist))

(defmacro reduce-prim-macro (proc! args alist)
   '(funcall (cbody ,proc!) ,args ,alist))
```

The three macros above are for primitives. The function below is for user defined functions. It normalizes the body of the closure in an environment that has been extended by matching the formal parameter pattern against the actual argument pattern. The actual argument may or may not have been normalized beforehand.

```
(defun expand-closure (proc! args!)
   (normalize (cbody proc!)
              (ext (calist proc!)
                   (match (cpattern proc!)
                          args!))))
```

2-Lisp–unlike Scheme–must analyze the left hand side of redexes first. The left hand side of a pair should normalize to a closure. After `reduce` has a closure, there are six cases to consider.

Cases 1 and 2. Both primitive exprs and user-defined exprs have their arguments normalized. The difference, from the point of view of `reduce`, is that primitive exprs get to manipulate `r`, the a-list that serves as the current environment.

Cases 3 and 4. Neither kind of impr has its arguments normalized. A user-defined impr is given a handle on the actual structure that is its argument. A primitive impr is given the actual structure. Remember, a primitive impr is actually Franz code. A handle would be nothing more than an extra cons-cell getting in the way. Things would be different if the either the primitive or `reduce` itself were written in 2-Lisp.

Cases 5 and 6. Once again, the arguments are left unnormalized. A user-defined macro is given a handle on the argument structure with which it was called. The macro body should designate the structure that is to replace the calling pair. Thus, it normalizes to a handle (the only normal-form structure that designates another structure). The structure this handle designates is then normalized. `reduce` knows that primitive macros don't bother to tack on the handle.

```
(defun reduce (proc args r)
   (let ([proc! (normalize proc r)])
        (caseq (ctype proc!)
           [expr  (let ([args! (normalize args r)])
                       (if (cprim? proc!)
                           (reduce-prim-expr proc! args! r)
                           (expand-closure proc! args!)))]
           [impr  (if (cprim? proc!)
                      (reduce-prim-impr proc! args r)
                         (expand-closure proc! (hcons args)))]
           [macro (if (cprim? proc!)
                      (normalize (reduce-prim-macro proc! args r) r)
                      (normalize (hstruc
                                    (expand-closure proc! (hcons args))) r))]
           [t (error "reduce: type not expr, impr, or macro")])))
```

7. f/alist.l. One a-list manipulator is defined as a function. The rest are defined as macros in `m/alist.l`. Notice especially the second `cond` clause of `match`. This is not at all what one would expect. When patterns are being matched against bindings, Smith breaks the rules for the sake of convenience. If the binding is `'[1 2 3]`, he allows it to be replaced by `['1 '2 '3]` if the latter will pattern-match better. This is used frequently by `imprs` and `macros`.

Notice also that if there is "too much binding, too little pattern" the extra arguments are ignored without an error. This can be argued for as a "feature." But in fact it was done for the speed that comes with minimal error-checking.

```
(defun match (pattern binding)
   (cond [(atom? pattern) (list (cons (asymbol pattern) binding))]
         [(and (hand? binding)
               (rail? (hstruc binding))) (match pattern (hr-to-rh binding))]
         [(rempty? pattern) nil]
         [(rempty? binding) (error "too much pattern, too little binding")]
         [t (nconc (match (rfirst pattern) (rfirst binding))
                   (match (rrest  pattern) (rrest  binding)))]))
```

8. f/primitives.l. This file contains the bodies of all the primitive closures. The bodies are taken out of the primitive closures so that they can be compiled. The one help function, **rh-to-hr**, is the inverse operation to **hr-to-rh**, defined in **m/alist.l**. It transforms a rail of handles to a handle on a rail. It is used once, in the **rcons** primitive.

```
(defun rh-to-hr (rail)
   (hcons (maprfirst '(lambda (hand) (hstruc hand))
                      rail)))

(def prim-numeral?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (nume? (hstruc struc)))))))

(def prim-boolean?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (bool? (hstruc struc)))))))

(def prim-closure?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (clos? (hstruc struc)))))))
```

```
(def prim-ccons
   (lambda (args! r)
      (let  ([type    (asymbol  (hstruc (rnth1 args!)))]
             [alist   (malist (hstruc (rnth2 args!)))]
             [pattern         (hstruc (rnth3 args!))]
             [body            (hstruc (rnth4 args!))])
         (hcons (mk-clos type alist pattern body)))))

(def prim-ctype
   (lambda (args! r)
      (let  ([clos (hstruc (rnth1 args!))])
         (hcons (mk-atom (ctype clos))))))

(def prim-cmap
   (lambda (args! r)
      (let* ([clos (hstruc (rnth1 args!))]
             [alist (calist clos)])
         (if (and (atom alist)
                  (not (null alist)))
             (hcons (mk-atom alist))
             (hcons (mk-map alist))))))

(def prim-cpattern
   (lambda (args! r)
      (hcons (cpattern (hstruc (rnth1 args!))))))

(def prim-cbody
   (lambda (args! r)
      (hcons (cbody (hstruc (rnth1 args!))))))

(def prim-rail?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (rail? (hstruc struc)))))))

(def prim-rcons
   (lambda (args! r)
      (rh-to-hr args!)))

(def prim-rfirst
   (lambda (args! r)
      (hcons (rfirst (hstruc (rnth1 args!))))))
```

```
(def prim-rrest
   (lambda (args! r)
      (hcons (rrest (hstruc (rnth1 args!))))))

(def prim-rnth
   (lambda (args! r)
      (hcons (rnth (nfixnum (rnth1 args!))
                   (hstruc  (rnth2 args!))))))

(def prim-rprep
   (lambda (args! r)
      (let  ([head (hstruc (rnth1 args!))]
             [tail (hstruc (rnth2 args!))])
         (hcons (rprep head tail)))))

(def prim-rlength
   (lambda (args! r)
      (mk-nume (rlength (hstruc (rnth1 args!))))))

(def prim-rset-n!
   (lambda (args! r)
      (hcons (rset-n! (nfixnum (rnth1 args!))
                 (hstruc  (rnth2 args!))
                 (hstruc  (rnth3 args!))))))

(def prim-rset-t!
   (lambda (args! r)
      (hcons (rset-t! (nfixnum (rnth1 args!))
                 (hstruc  (rnth2 args!))
                 (hstruc  (rnth3 args!))))))

(def prim-handle?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (hand? (hstruc struc)))))))

(def prim-atom?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (atom? (hstruc struc)))))))

(def prim-acons
   (lambda (args! r) (hcons (mk-atom (gensym)))))
```

```
(def prim-pair?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (pair? (hstruc struc)))))))

(def prim-pcons
   (lambda (args! r)
      (let  ([a (hstruc (rnth1 args!))]
             [d (hstruc (rnth2 args!))])
         (hcons (pcons a d)))))

(def prim-pcar
   (lambda (args! r)
      (let  ([p (hstruc (rnth1 args!))])
         (hcons (pcar p)))))

(def prim-pcdr
   (lambda (args! r)
      (let  ([p (hstruc (rnth1 args!))])
         (hcons (pcdr p)))))

(def prim-pset-car!
   (lambda (args! r)
      (hcons (pset-car! (hstruc (rnth1 args!))
                (hstruc (rnth2 args!))))))

(def prim-pset-cdr!
   (lambda (args! r)
      (hcons (pset-cdr! (hstruc (rnth1 args!))
                (hstruc (rnth2 args!))))))

(def prim-map?
   (lambda (args! r)
      (let ([struc (rnth1 args!)])
         (mk-bool (and (hand? struc)
                       (map? (hstruc struc)))))))

(def prim-mcons
   (lambda (args! r)
      (let ([newer-map (hstruc (rnth1 args!))]
            [older-map (hstruc (rnth2 args!))])
         (hcons (mcons newer-map older-map)))))
```

```
(def prim-mbound?
   (lambda (args! r)
      (let ([atom (hstruc (rnth1 args!))]
            [map (hstruc (rnth2 args!))])
         (mk-bool (lookup atom (malist map))))))

(def prim-mbinding
   (lambda (args! r)
      (let ([atom (hstruc (rnth1 args!))]
            [map  (hstruc (rnth2 args!))])
         (hcons (binding atom (malist map))))))

(def prim-mbind
   (lambda (args! r)
      (let ([pattern (hstruc (rnth1 args!))]
            [binding (hstruc (rnth2 args!))]
            [map     (hstruc (rnth3 args!))])
         (hcons (mk-map (bind pattern
                              binding
                              (malist map)))))))

(def prim-mset-binding!
   (lambda (args! r)
      (let ([atom    (hstruc (rnth1 args!))]
            [binding (hstruc (rnth2 args!))]
            [map     (hstruc (rnth3 args!))])
         (set-binding! atom binding (malist map))
         (hcons binding))))

(def prim-number?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (nume? struc)))))

(def prim-+
   (lambda (args! r)
      (let  ([n1 (nfixnum (rnth1 args!))]
             [n2 (nfixnum (rnth2 args!))])
         (mk-nume (+ n1 n2)))))

(def prim--
   (lambda (args! r)
      (let  ([n1 (nfixnum (rnth1 args!))]
             [n2 (nfixnum (rnth2 args!))])
         (mk-nume (- n1 n2)))))
```

```
(def prim-*
   (lambda (args! r)
      (let  ([n1 (nfixnum (rnth1 args!))]
             [n2 (nfixnum (rnth2 args!))])
         (mk-nume (* n1 n2)))))

(def prim-/
   (lambda (args! r)
      (let  ([n1 (nfixnum (rnth1 args!))]
             [n2 (nfixnum (rnth2 args!))])
         (mk-nume (/ n1 n2)))))

(def prim-truth-value?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (bool? struc)))))

(def prim-and
   (lambda (args! r)
      (mk-bool (and (btnil (rnth1 args!))
                    (btnil (rnth2 args!))))))

(def prim-or
   (lambda (args! r)
      (mk-bool (or (btnil (rnth1 args!))
                   (btnil (rnth2 args!))))))

(def prim-not
   (lambda (args! r)
      (mk-bool (not (btnil (rnth1 args!))))))

(def prim-sequence?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (rail? struc)))))

(def prim-scons
   (lambda (args! r) args!))

(def prim-sfirst
   (lambda (args! r)
      (rfirst (rnth1 args!))))
```

```
(def prim-srest
   (lambda (args! r)
      (rrest (rnth1 args!))))

(def prim-snth
   (lambda (args! r)
      (rnth (nfixnum (rnth1 args!)) (rnth2 args!))))

(def prim-sprep
   (lambda (args! r)
      (rprep (rnth1 args!) (rnth2 args!))))

(def prim-slength
   (lambda (args! r)
      (mk-nume (rlength (rnth1 args!)))))

(def prim-function?
   (lambda (args! r)
      (let  ([struc (rnth1 args!)])
         (mk-bool (clos? struc)))))

(def prim-lambda
   (lambda (args r)
      (let* ([type    (asymbol (rnth1 args))]
             [pattern        (rnth2 args) ]
             [body           (rnth3 args) ])
         (mk-clos type r pattern body))))

(def prim-xcons
   (lambda (args! r)
      (hcons (pcons (hstruc (rfirst args!))
                    (maprfirst '(lambda (hand) (hstruc hand))
                       (rrest args!))))))
```

=? says that it can't deal with functions, but it makes an attempt nonetheless. It's true that it's impossible to tell in all cases whether two algorithms compute the same function. But that doesn't mean that it's never possible to tell. Similarly, if two maps are the same, they must designate the same environment.

```
(def prim-=?
   (lambda (args! r)
      (let ([s1 (rnth1 args!)]
            [s2 (rnth2 args!)])
         (mk-bool
            (cond [(hand? s1)
                   (and (hand? s2)
                        (2-equal? (hstruc s1)
                           (hstruc s2)))]
                  [(clos? s1)
                   (and (clos? s2)
                        (or (equal s1 s2)
                            (error "=? can't deal with functions")))]
                  [(map? s1)
                   (and (map? s2)
                        (or (equal s1 s2)
                            (error "=? can't deal with maps")))]
                  [t (equal s1 s2)])))))
```

```
(def prim-if
   (lambda (args r)
      (let* ([pred (rnth1 args)]
             [then (rnth2 args)]
             [else (rnth3 args)]
             [pred! (normalize pred r)])
         (if (btnil pred!)
             (normalize then r)
             (normalize else r)))))
```

The begin macro is primitive because Franz code is faster than 2-Lisp code, and begin is frequently used.

```
(def prim-begin
   (lambda (args r)
      (if (runit? args)
          (rlast args)
          (xcons (xcons (mk-atom 'lambda)
                        (mk-atom 'expr)
                        (mk-atom '?)
                        (rlast args))
                 (pcons (mk-atom 'begin)
                        (rstart args))))))
```

`up` just tacks on a handle. Notice that exprs are supposed to be defined in terms of the designations of the arguments that are typed. But of course when the boolean `$T` is typed in, the Franz code manipulates `(B . t)`, not *Truth*.

```
(def prim-up
   (lambda (args! r)
      (let  ([x (rnth1 args!)])
         (hcons x))))

(def prim-down
   (lambda (args! r)
      (let ([hand (rnth1 args!)])
         (if (hand? hand)
             (normalize (hstruc hand) r)
             (error "error in down")))))

(def prim-define!
   (lambda (args r)
      (let* ([id    (rnth1 args)]
             [clos  (rnth2 args)]
             [clos! (normalize clos r)])
         (progn (set-binding! id clos! r)
                (hcons id)))))

(def prim-set!
   (lambda (args r)
      (let* ([id    (rnth1 args)]
             [bind  (rnth2 args)]
             [bind! (normalize bind r)])
         (progn (set-binding! id bind! r)
                bind!))))
```

```
(def prim-read!
   (lambda (args! r)
      (hcons (2-read))))

(def prim-print!
   (lambda (args! r)
      (progn (2-print (hstruc (rnth1 args!)))
             (drain)
             (mk-bool t))))

(def prim-new-line!
   (lambda (args r)
      (progn (terpri)
             (mk-bool t))))
```

9. f/initalist.l. This file contains one help function and the initial association list. The help function, `symbols-to-ra`, takes any number of unquoted Franz symbols and creates a rail of 2-Lisp atoms. It is used to make the pattern part of each primitive closure. For primitives, the pattern is just for show. It's there for when the primitive closure is printed. The real formal parameter list is part of the Franz-code body of the primitive.

```
(def symbols-to-ra
   (macro (l)
          (let ([symbols (cdr l)])
               (cons 'rcons
                  (mapcar '(lambda (symbol) `(mk-atom (quote ,symbol)))
                     symbols)))))
```

And now, the initial association list.

```
(setq initalist
   `(  [numeral? . ,(mk-clos 'expr 'prim
                          [symbols-to-ra arg]
                          'prim-numeral?)]

       [boolean? . ,(mk-clos 'expr 'prim
                          [symbols-to-ra arg]
                          'prim-boolean?)]
```

```
[closure? . ,(mk-clos 'expr 'prim
                 [symbols-to-ra arg]
                 'prim-closure?)]

[ccons . ,(mk-clos 'expr 'prim
              [symbols-to-ra atom map pattern body]
              'prim-ccons)]

[ctype . ,(mk-clos 'expr 'prim
              [symbols-to-ra clos]
              'prim-ctype)]

[cmap . ,(mk-clos 'expr 'prim
             [symbols-to-ra clos]
             'prim-cmap)]

[cpattern . ,(mk-clos 'expr 'prim
                 [symbols-to-ra clos]
                 'prim-cpattern)]

[cbody . ,(mk-clos 'expr 'prim
              [symbols-to-ra clos]
              'prim-cbody)]

[rail? . ,(mk-clos 'expr 'prim
              [symbols-to-ra arg]
              'prim-rail?)]

[rcons . ,(mk-clos 'expr 'prim
              [symbols-to-ra struc1 struc2 |...|]
              'prim-rcons)]

[rfirst . ,(mk-clos 'expr 'prim
               [symbols-to-ra rail]
               'prim-rfirst)]

[rrest . ,(mk-clos 'expr 'prim
              [symbols-to-ra rail]
              'prim-rrest)]

[rnth . ,(mk-clos 'expr 'prim
             [symbols-to-ra n rail]
             'prim-rnth)]
```

```
[rprep . ,(mk-clos 'expr 'prim
                   [symbols-to-ra head tail]
                   'prim-rprep)]

[rlength . ,(mk-clos 'expr 'prim
                     [symbols-to-ra rail]
                     'prim-rlength)]

[rset-n! . ,(mk-clos 'expr 'prim
                     [symbols-to-ra n rail struc]
                     'prim-rset-n!)]

[rset-t! . ,(mk-clos 'expr 'prim
                     [symbols-to-ra n rail tail]
                     'prim-rset-t!)]

[handle? . ,(mk-clos 'expr 'prim
                     [symbols-to-ra arg]
                     'prim-handle?)]

[atom? . ,(mk-clos 'expr 'prim
                   [symbols-to-ra arg]
                   'prim-atom?)]

[acons . ,(mk-clos 'expr 'prim
                   [symbols-to-ra ]
                   'prim-acons)]

[pair? . ,(mk-clos 'expr 'prim
                   [symbols-to-ra arg]
                   'prim-pair?)]

[pcons . ,(mk-clos 'expr 'prim
                   [symbols-to-ra car cdr]
                   'prim-pcons)]

[pcar . ,(mk-clos 'expr 'prim
                  [symbols-to-ra pair]
                  'prim-pcar)]

[pcdr . ,(mk-clos 'expr 'prim
                  [symbols-to-ra pair]
                  'prim-pcdr)]
```

```
[pset-car! . ,(mk-clos 'expr 'prim
                  [symbols-to-ra pair car]
                  'prim-pset-car!)]

[pset-cdr! . ,(mk-clos 'expr 'prim
                  [symbols-to-ra pair cdr]
                  'prim-pset-cdr!)]

[map? . ,(mk-clos 'expr 'prim
             [symbols-to-ra arg]
             'prim-map?)]

[mcons . ,(mk-clos 'expr 'prim
              [symbols-to-ra newer-map older-map]
              'prim-mcons)]

[mbound? . ,(mk-clos 'expr 'prim
                [symbols-to-ra atom map]
                'prim-mbound?)]

[mbinding . ,(mk-clos 'expr 'prim
                 [symbols-to-ra atom map]
                 'prim-mbinding)]

[mbind . ,(mk-clos 'expr 'prim
              [symbols-to-ra pattern binding map]
              'prim-mbind)]

[mset-binding! . ,(mk-clos 'expr 'prim
                      [symbols-to-ra atom struc map]
                      'prim-mset-binding!)]

[number? . ,(mk-clos 'expr 'prim
                [symbols-to-ra arg]
                'prim-number?)]

[+ . ,(mk-clos 'expr 'prim
          [symbols-to-ra n1 n2]
          'prim-+)]

[- . ,(mk-clos 'expr 'prim
          [symbols-to-ra n1 n2]
          'prim--)]
```

```
[* . ,(mk-clos 'expr 'prim
          [symbols-to-ra n1 n2]
          'prim-*)]

[/ . ,(mk-clos 'expr 'prim
          [symbols-to-ra n1 n2]
          'prim-/)]

[truth-value? . ,(mk-clos 'expr 'prim
                     [symbols-to-ra arg]
                     'prim-truth-value?)]

[and . ,(mk-clos 'expr 'prim
           [symbols-to-ra tv1 tv2]
           'prim-and)]

[or  . ,(mk-clos 'expr 'prim
           [symbols-to-ra tv1 tv2]
           'prim-or)]

[not . ,(mk-clos 'expr 'prim
           [symbols-to-ra tv]
           'prim-not)]

[sequence? . ,(mk-clos 'expr 'prim
                 [symbols-to-ra arg]
                 'prim-sequence?)]

[scons  . ,(mk-clos 'expr 'prim
              [symbols-to-ra e1 e2 |...|]
              'prim-scons)]

[sfirst . ,(mk-clos 'expr 'prim
              [symbols-to-ra sequ]
              'prim-sfirst)]

[srest . ,(mk-clos 'expr 'prim
             [symbols-to-ra sequ]
             'prim-srest)]

[snth . ,(mk-clos 'expr 'prim
            [symbols-to-ra n sequ]
            'prim-snth)]
```

```
[sprep . ,(mk-clos 'expr 'prim
             [symbols-to-ra first sequ]
             'prim-sprep)]

[slength . ,(mk-clos 'expr 'prim
               [symbols-to-ra sequ]
               'prim-slength)]

[function? . ,(mk-clos 'expr 'prim
                 [symbols-to-ra arg]
                 'prim-function?)]

[lambda . ,(mk-clos 'impr 'prim
              [symbols-to-ra type pattern body]
              'prim-lambda)]

[xcons . ,(mk-clos 'expr 'prim
             [symbols-to-ra proc arg1 arg2 |...|]
             'prim-xcons)]

[=?  . ,(mk-clos 'expr 'prim
           [symbols-to-ra arg1 arg2]
           'prim-=?)]

[if . ,(mk-clos 'impr 'prim
          [symbols-to-ra pred then else]
          'prim-if)]

[begin . ,(mk-clos 'macro 'prim
             [symbols-to-ra exp1 exp2 |...|]
             'prim-begin)]

[up . ,(mk-clos 'expr 'prim
          [symbols-to-ra signified]
          'prim-up)]

[down . ,(mk-clos 'expr 'prim
            [symbols-to-ra sign]
            'prim-down)]

[define! . ,(mk-clos 'impr 'prim
               [symbols-to-ra id binding]
               'prim-define!)]
```

```
       [set! . ,(mk-clos 'impr 'prim
                   [symbols-to-ra id binding]
                   'prim-set!)]

       [read! . ,(mk-clos 'impr 'prim
                    [symbols-to-ra]
                    'prim-read!)]

       [print! . ,(mk-clos 'expr 'prim
                     [symbols-to-ra struc]
                     'prim-print!)]

       [new-line! . ,(mk-clos 'impr 'prim
                        [symbols-to-ra]
                        'prim-new-line!)]
       ))
```

10. f/loop.l. This file contains the top level `read-normalize-print` loop. `2-Lisp` serves two purposes. First, it provides a convenient entry point; it's the intended entry point. Second, it uses `errset` to recover from errors without throwing the user back to Franz Lisp.

```
(defun 2-Lisp ()
   (prog nil
      repeat
      (if (errset (read-normalize-print initalist))
          (return 'stopped))
      (go repeat)))
```

The most useful and interesting aspect of `read-normalize-print` is the override facility. Overrides are typed in preceded with a tilde, as in `~dot`. They are represented internally like this: `(0 . dot)`. They normalize to themselves, and are printed out like this: `Override: dot`. After they are printed, but before `read-normalize-print` cycles back to read the next input, they are matched against a list of meaningful overrides. The meaningful overrides are `~read`, `~echo`, `~print`, `~dot`, `~stop` and `~exit`. If there is a match, the corresponding switch is toggled (`~read`, `~echo`, `~print`, or `~dot`) or the 2-Lisp processor returns control to Franz Lisp and its readtable (`~stop` or `~exit`).

```
(defun read-normalize-print (r)
   (prog (in out)
         (setq read-flag  nil)
         (setq echo-flag  nil)
         (setq print-flag nil)
         (setq dot-flag   nil)
         (setq readtable 2-readtable)
    loop (setq in (prompt&read))
         (setq out (if echo-flag
                       in
                       (normalize in r)))
         (prompt&reply out)
         (if (and (not (atom out))
                  (over? out))
             (selectq (ofranz out)
                [read   (setq read-flag (not read-flag))
                        (setq readtable (if read-flag
                                            Franz-readtable
                                            2-readtable))]
                [echo   (setq echo-flag  (not echo-flag))]
                [print  (setq print-flag (not print-flag))]
                [dot    (setq dot-flag (not dot-flag))]
                [stop   (progn (setq readtable Franz-readtable)
                               (return 'stopped))]
                [exit   (progn (setq readtable Franz-readtable)
                               (return 'stopped))]))
         (go loop)))
```

Notice that a garbage collection is done before reading, but after the user has been prompted. The **drain** makes sure that the prompt is printed right away, not just before the **read**.

```
(defun prompt&read ()
   (progn (if read-flag
              (patom '|1-read ?  |)
              (patom '|2-Lisp ?  |))
          (drain)
          (gc)
          (if read-flag
              (read)
              (2-read*))))
```

```
(defun prompt&reply (out)
   (progn (if echo-flag
             (patom '|       =  |)
             (patom '|     ==>  |))
          (if print-flag
             (1-print out)
             (2-print out))
          (terpri)))
```

These two I/O functions should make clear the effects of the `~read` and `~print` overrides. The code for `read-normalize-print` shows what `~echo`, `~stop` and `~exit` do. The `dot-flag` is referenced in `f/write.1`.

6. CONCLUSIONS

The overall goal of convenience for the user is achieved by making this implementation conform closely to the special syntax presented in Smith [84]. The primitives are named uniformly and descriptively. Design choices are uniformly decided in favor of speed over space. The Franz code is written in as natural a style as possible to make clear exactly how the processor does what it does; issues of tail recursion are ignored.

7. ACKNOWLEDGEMENT

This project was directed by Daniel Friedman.

8. REFERENCES

[Foderaro *et al.* 83]
Foderaro, J.K., Sklower, K.L., and Layer, K., "The Franz Lisp Manual," University of California at Berkeley (June 1983).

[Friedman *et al.* 84]
Friedman, D.P., Haynes, C.T., Kohlbecker, E., and Wand, M., "The Scheme 84 Reference Manual," Indiana University Computer Science Department Technical Report No. 153 (March 1984).

[Kohlbecker 84]
Kohlbecker, E., "Using mkmac," Indiana University Computer Science Department Technical Report No. 157 (March 1984).

[Smith 82]
Smith, B.C., *Reflection and Semantics in a Procedural Language*, MIT/LCS/TR-272, Mass. Inst. of Tech., Cambridge, MA, January, 1982.

[Smith 84]
Smith, B.C., "Reflection and Semantics in Lisp," *Conf. Rec. 11th ACM Symp. on Principles of Programming Languages* (1984).

Appendix—Names of Primitives

The following table shows the correspondence between Smith's names for primitives and the names used in this implementation.

Smith		This implementation	
NUMERAL		numeral?	
BOOLEAN		boolean?	
CLOSURE		closure?	
	CCONS		ccons
	PROCEDURE-TYPE		ctype
	ENVIRONMENT		cmap
	PATTERN		cpattern
	BODY		cbody
RAIL		rail?	
	RCONS		rcons
			rfirst
			rrest
			rnth
			rprep
			rlength
	RPLACN		rset-n!
	RPLACT		rset-t!
HANDLE		handle?	
ATOM		atom?	
	ACONS		acons
PAIR		pair?	
	PCONS		pcons
	CAR		pcar
	CDR		pcdr
	RPLACA		pset-car!
	RPLACD		pset-cdr!
		map?	
			mcons
			mbound?
			mbinding
			mbind
			mset-binding!

NUMBER	number?
+	+
-	-
*	*
/	/
TRUTH-VALUE	truth-value?
AND	and
OR	or
NOT	not
FUNCTION	function?
LAMBDA	lambda
EXPR	expr
IMPR	impr
MACRO	macro
SEQUENCE	sequence?
SCONS	scons
	sfirst
	srest
	snth
	sprep
	slength
vectors	rails and sequences
1ST	
REST	
NTH	
PREP	
LENGTH	
redexes	redexes
XCONS	xcons

=		**=?**	
IF		**if**	
BLOCK		**begin**	
NAME		**up**	
	↑		^
REFERENT		**down**	
	↓		_
DEFINE		**define!**	
SET		**set!**	
READ		**read!**	
PRINT		**print!**	
TERPRI		**new-line!**	

Appendix—Quick Guide

This appendix is intended as a supplement to the previous appendix. Together, the two appendices serve as a quick guide-sheet to this implementation of 2-Lisp.

on switch	`-> (load "/usiu/charlie/2/s/speed.1")` `-> (2-Lisp)`	
off switch	`~stop`	`(0 . stop)`
1-read/2-read toggle	`~read`	`(0 . read)`
echo/normalize toggle	`~echo`	`(0 . echo)`
1-print/2-print toggle	`~print`	`(0 . print)`
dot/not toggle	`~dot`	`(0 . dot)`
return to Franz readtable	`<2>:  ~~`	