19. The LOOP Iteration Macro

19.1 Introduction

loop is a Lisp macro that provides a programmable iteration facility. The same loop module operates compatibly in Zetalisp, Maclisp (PDP-10 and Multics), and NIL, and a moderately compatible package is under development for the MDL programming environment. loop was inspired by the FOR facility of CLISP in InterLisp; however, it is not compatible and differs in several details.

The general approach is that a form introduced by the word loop generates a single program loop, into which a large variety of features can be incorporated. The loop consists of some initialization (*prologue*) code, a body that may be executed several times, and some exit (*epilogue*) code. Variables may be declared local to the loop. The special features of loop are concerned with loop variables, deciding when to end the iteration, putting user-written code into the loop, returning a value from the construct, and iterating a variable through various real or virtual sets of values.

The loop form consists of a series of clauses, each introduced by a "keyword" symbol. These symbols are keywords from loop's point of view; they are not keywords in the usual sense (symbols in the keyword package). loop ignores the package when it compares a symbol against the known keywords.

Forms appearing in or implied by the clauses of a loop form are classed as those to be executed as initialization code, body code, and/or exit code; within each part of the template filled in by loop, they are executed strictly in the order implied by the original composition. Thus, just as in ordinary Lisp code, side-effects may be used, and one piece of code may depend on following another for its proper operation. This is the principal philosophic difference from InterLisp's FOR facility.

Note that loop forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to do.

Here are some examples to illustrate the use of loop.

```
(defun print-elements-of-list (list-of-elements)
    (loop for element in list-of-elements
          do (print element)))
```

prints each element in its argument, which should be a list. It returns nil.

```
(bar (defun gather-alist-entries (list-of-pairs)
       (loop for pair in list-of-pairs
             collect (car pair)))
```

takes an association list and returns a list of the keys; that is, (gather-alist-entries '((foo 1 2) (bar 259) (baz))) returns (foo bar baz).

```
(defun extract-interesting-numbers (start-value end-value)
  (loop for number from start-value to end-value
        when (interesting-p number) collect number))
```

takes two arguments, which should be integers, and returns a list of all the numbers in that range (inclusive) which satisfy the predicate interesting-p.

```
(defun find-maximum-element (an-array)
  (loop for i from 0 below (array-dimension-n 1 an-array)
        maximize (aref an-array i)))
```

returns the maximum of the elements of its argument, a one-dimensional array.

```
(defun my-remove (object list)
  (loop for element in list
        unless (equal object element) collect element))
```

is like the standard function remove, except that it copies the entire list.

```
(defun find-frob (list)
  (loop for element in list
        when (frobp element) return element
        finally (ferror nil "No frob found in the list ~S"
                        list)))
```

returns the first element of its list argument which satisfies the predicate frobp. If none is found, an error is signaled.

Common Lisp defines loop as equivalent to do-forever: it is used with a body consisting only of forms to be evaluated until a nonlocal exit happens. This is incompatible with the traditional loop macro which this chapter is about. However, it is possible to tell which meaning of loop the programmer intended: in the traditional loop macro, it must be a symbol, while in the Common Lisp loop it is useless to use a symbol there. Therefore, if the first argument of a loop form is not a symbol, it is treated as a Common Lisp loop.

19.2 Clauses

Internally, loop constructs a prog which includes variable bindings, pre-iteration (initialization) code, post-iteration (exit) code, the body of the iteration, and stepping of variables of iteration to their next values (which happens on every iteration after the body is executed).

A *clause* consists of a keyword symbol and any Lisp forms and keywords that it deals with. For example,

```
(loop for x in l do (print x)),
```

contains two clauses, `for x in l` and `do (print x)`. Certain of the parts of the clause will be described as being *expressions*, e.g. (print x) in the above. An expression can be a single Lisp

form, or a series of forms implicitly collected with **progn**. An expression is terminated by the next following atom, which is taken to be a keyword. This syntax allows only the first form in an expression to be atomic, but makes misspelled keywords more easily detectable.

loop uses print-name equality to compare keywords so that **loop** forms may be written without package prefixes; in Lisp implementations that do not have packages, **eq** is used for comparison.

Bindings and iteration variable steppings may be performed either sequentially or in parallel. This affects how the stepping of one iteration variable may depend on the value of another. The syntax for distinguishing the two will be described with the corresponding clauses. When a set of variables are to be bound in parallel, all of the initial values are computed and then all the bindings are established. Subsequent bindings will be performed inside of that binding environment. When the same variables are stepped, all the new values are computed and then the variables are set.

19.2.1 Iteration-Driving Clauses

These clauses all create a *variable of iteration*, which is bound locally to the loop and takes on a new value on each successive iteration. Note that if more than one iteration-driving clause is used in the same loop, several variables are created that all step together through their values; when any of the iterations terminates, the entire loop terminates. Nested iterations are not generated; for those, you need a second **loop** form in the body of the loop. In order not to produce strange interactions, iteration-driving clauses are required to precede any clauses that produce body code: that is, all except those that produce prologue or epilogue code (**initially** and **finally**), bindings (**with**), the **named** clause, and the iteration termination clauses (**while** and **until**).

Clauses which drive the iteration may be arranged to perform their testing and stepping either in series or in parallel. They are by default grouped in series, which allows the stepping computation of one clause to use the just-computed values of the iteration variables of previous clauses. They may be made to step in parallel, as is the case with the **do** special form, by "joining" the iteration clauses with the keyword **and**. The form this typically takes is something like

```
(loop ... for x = (f) and for y = init then (g x) ...)
```

which sets **x** to **(f)** on every iteration, and binds **y** to the value of *init* for the first iteration, and on every iteration thereafter sets it to **(g x)**, where **x** still has the value from the *previous* iteration. Thus, if the calls to **f** and **g** are not order-dependent, this would be best written as

```
(loop ... for y = init then (g x) for x = (f) ...)
```

because, as a general rule, parallel stepping has more overhead than sequential stepping. Similarly, the example

```
(loop for sublist on some-list
      and for previous = 'undefined then sublist
      ...)
```

which is equivalent to the **do** construct

```
(do ((sublist some-list (cdr sublist))
     (previous 'undefined sublist))
    ((null sublist) ...)
  ...)
```

in terms of stepping, would be better written as

```
(loop for previous = 'undefined then sublist
      for sublist on some-list
      ...)
```

When iteration-driving clauses are joined with **and**, if the token following the **and** is not a keyword that introduces an iteration driving clause, it is assumed to be the same as the keyword that introduced the most recent clause; thus, the above example showing parallel stepping could have been written as

```
(loop for sublist on some-list
      and previous = 'undefined then sublist
      ...)
```

The order of evaluation in iteration-driving clauses is as follows: those expressions that are only evaluated once are evaluated in order at the beginning of the form, during the variable-binding phase, while those expressions that are evaluated each time around the loop are evaluated in order in the body.

One common and simple iteration-driving clause is **repeat**:

repeat *expression*

Evaluates *expression* (during the variable binding phase), and causes the **loop** to iterate that many times. *expression* is expected to evaluate to an integer. If *expression* evaluates to a zero or negative result, the body code will not be executed.

All remaining iteration-driving clauses are subdispatches of the keyword **for**, which is synonomous with **as**. In all of them a *variable of iteration* is specified. Note that, in general, if an iteration-driving clause implicitly supplies an endtest, the value of this iteration variable is undefined as the loop is exited (i.e., when the epilogue code is run). This is discussed in more detail in section 19.5.

Here are all of the varieties of **for** clauses. Optional parts are enclosed in curly brackets.

for *var* **in** *expr1* {**by** *expr2*}

Iterates over each of the elements in the list *expr1*. If the **by** subclause is present, *expr2* is evaluated once on entry to the loop to supply the function to be used to fetch successive sublists, instead of **cdr**.

for *var* **on** *expr1* {**by** *expr2*}

Like the previous **for** format, except that *var* is set to successive sublists of the list instead of successive elements. Note that **loop** uses a **null** rather than an **atom** test to implement both this and the preceding clause.

for *var* **=** *expr*

On each iteration, *expr* is evaluated and *var* is set to the result.

for *var* **=** *expr1* **then** *expr2*

var is bound to *expr1* when the loop is entered, and set to *expr2* (re-evaluated) at all

but the first iteration. Since *expr1* is evaluated during the binding phase, it cannot reference other iteration variables set before it: for that, use the following:

for *var* first *expr1* then *expr2*

Sets *var* to *expr1* on the first iteration, and to *expr2* (re-evaluated) on each succeeding iteration. The evaluation of both expressions is performed *inside* of the loop binding environment, before the loop body. This allows the first value of *var* to come from the first value of some other iteration variable, allowing such constructs as

```
(loop for term in poly
      for ans first (car term)
              then (gcd ans (car term))
      finally (return ans))
```

for *var* from *expr1* {to *expr2*} {by *expr3*}

This performs numeric iteration. *var* is initialized to *expr1*, and on each succeeding iteration is incremented by *expr3* (default 1). If the to phrase is given, the iteration terminates when *var* becomes greater than *expr2*. Each of the expressions is evaluated only once, and the to and by phrases may be written in either order. Alternative keywords may be used in place of to; this choice controls the direction of stepping and the step at which the loop terminates. downto instead of to says that *var* is decremented by the step value, and the endtest is adjusted accordingly. If below is used instead of to, or above instead of downto, the iteration terminates before *expr2* is reached, rather than after. Note that the to variant appropriate for the direction of stepping must be used for the endtest to be formed correctly; i.e. the code will not work if *expr3* is negative or zero. If no limit-specifying clause is given, then the direction of the stepping may be specified as decreasing by using downfrom instead of from. upfrom may also be used instead of from; it forces the stepping direction to be increasing.

for *var* being *expr* and its *path* ...

for *var* being {each|the} *path* ...

Provides a user-definable iteration facility. *path* names the manner in which the iteration is to be performed. The ellipsis indicates where various path dependent preposition/expression pairs may appear. See the section on Iteration Paths (page 363) for complete documentation.

19.2.2 Bindings

The with keyword may be used to establish initial bindings, that is, variables that are local to the loop but are only set once, rather than on each iteration. The with clause looks like:

with *var1* {= *expr1*}
{and *var2* {= *expr2*}}...

If no *expr* is given, the variable is initialized to nil.

with bindings linked by and are performed in parallel; those not linked are performed sequentially. That is,

```
(loop with a = (foo) and b = (bar) and c
      ...)
```

binds the variables like

```
(let ((a (foo)) (b (bar)) c) ...)
```
whereas
```
(loop with a = (foo) with b = (bar a) with c ...)
```
binds the variables like
```
(let ((a (foo)))
  (let ((b (bar)))
    (let (c) ...)))
```
All *expr*'s in with clauses are evaluated in the order they are written, in lambda expressions surrounding the generated prog. The loop expression
```
(loop with a = xa and b = xb
      with c = xc
      for d = xd then (f d)
        and e = xe then (g e d)
      for p in xp
      with q = xq
      ...)
```
produces the following binding contour, where t1 is a loop-generated temporary:
```
(let ((a xa) (b xb))
  (let ((c xc))
    (let ((d xd) (e xe))
      (let ((p nil) (t1 xp))
        (let ((q xq))
          ...)))))
```
Because all expressions in with clauses are evaluated during the variable binding phase, they are best placed near the front of the loop form for stylistic reasons.

For binding more than one variable with no particular initialization, one may use the construct

with *variable-list* {and ...}

as in
```
with (i j k t1 t2) ...
```
These are cases of *destructuring* which loop handles specially; destructuring and data type keywords are discussed in section 19.4.

19.2.3 Entrance and Exit

initially *expression*

Puts *expression* into the *prologue* of the iteration. It will be evaluated before any other initialization code except for initial bindings. For the sake of good style, the initially clause should therefore be placed after any with clauses but before the main body of the loop.

finally *expression*

Puts *expression* into the *epilogue* of the loop, which is evaluated when the iteration terminates (other than by an explicit return). For stylistic reasons, then, this clause should appear last in the loop body. Note that certain clauses may generate code which terminates the iteration without running the epilogue code; this behavior is noted with those clauses. Most notable of these are those described in the section

19.2.7, Aggregated Boolean Tests. This clause may be used to cause the loop to return values in a non-standard way:

```
(loop for n in l
      sum n into the-sum
      count t into the-count
      finally (return (quotient the-sum the-count)))
```

19.2.4 Side Effects

do *expression*
doing *expression*

expression is evaluated each time through the loop, as shown in the print-elements-of-list example on page 350.

19.2.5 Values

The following clauses accumulate a return value for the iteration in some manner. The general form is

type-of-collection expr {into *var*}

where *type-of-collection* is a loop keyword, and *expr* is the thing being accumulated somehow. If no into is specified, then the accumulation will be returned when the loop terminates. If there is an into, then when the epilogue of the loop is reached, *var* (a variable automatically bound locally in the loop) will have been set to the accumulated result and may be used by the epilogue code. In this way, a user may accumulate and somehow pass back multiple values from a single loop, or use them during the loop. It is safe to reference these variables during the loop, but they should not be modified until the epilogue code of the loop is reached. For example,

```
(loop for x in list
      collect (foo x) into foo-list
      collect (bar x) into bar-list
      collect (baz x) into baz-list
     finally (return (list foo-list bar-list baz-list)))
```

has the same effect as

```
(do ((#:g0001 list (cdr #:g0001))
     (x) (foo-list) (bar-list) (baz-list))
    ((null #:g0001)
     (list (nreverse foo-list)
           (nreverse bar-list)
           (nreverse baz-list)))
  (setq x (car #:g0001))
  (setq foo-list (cons (foo x) foo-list))
  (setq bar-list (cons (bar x) bar-list))
  (setq baz-list (cons (baz x) baz-list)))
```

except that loop arranges to form the lists in the correct order, obviating the nreverses at the end, and allowing the lists to be examined during the computation. (This is how the expression would print; this text would not read in properly because a new uninterned symbol would be created by each use of #:.)

collect *expr* {into *var*}
collecting ...

Causes the values of *expr* on each iteration to be collected into a list.

nconc *expr* {into *var*}
nconcing ...
append ...
appending ...

Like collect, but the results are nconc'ed or append'ed together as appropriate.

```
(loop for i from 1 to 3
      nconc (list i (* i i)))
  => (1 1 2 4 3 9)
```

count *expr* {into *var*}
counting ...

If *expr* evaluates non-nil, a counter is incremented.

sum *expr* {into *var*}
summing ...

Evaluates *expr* on each iteration and accumulates the sum of all the values.

maximize *expr* {into *var*}
minimize ...

Computes the maximum (or minimum) of *expr* over all iterations.
Note that if the loop iterates zero times, or if conditionalization prevents the code of this clause from being executed, the result will be meaningless.

Not only may there be multiple accumulations in a loop, but a single accumulation may come from multiple places *within the same* loop *form*. Obviously, the types of the collection must be compatible. collect, nconc, and append may all be mixed, as may sum and count, and maximize and minimize. For example,

```
(loop for x in '(a b c) for y in '((1 2) (3 4) (5 6))
      collect x
      append y)
  => (a 1 2 b 3 4 c 5 6)
```

The following computes the average of the entries in the list *list-of-frobs*:

```
(loop for x in list-of-frobs
      count t into count-var
      sum x into sum-var
      finally (return (cli:// sum-var count-var)))
```

19.2.6 Endtests

The following clauses may be used to provide additional control over when the iteration gets terminated, possibly causing exit code (due to **finally**) to be performed and possibly returning a value (e.g., from **collect**).

while *expr*
: If *expr* evaluates to **nil**, the loop is exited, performing exit code (if any) and returning any accumulated value. The test is placed in the body of the loop where it is written. It may appear between sequential **for** clauses.

until *expr*
: Identical to **while (not** *expr*).

This may be needed, for example, to step through a strange data structure, as in

```
(loop until (top-of-concept-tree? concept)
      for concept = expr then (superior-concept concept)
      ...)
```

Note that the placement of the **until** clause before the **for** clause is valid in this case because of the definition of this particular variant of **for**, which *binds* **concept** to its first value rather than setting it from inside the **loop**.

The following may also be of use in terminating the iteration:

loop-finish *Macro*
: (loop-finish) causes the iteration to terminate "normally", like implicit termination by an iteration-driving clause, or by the use of **while** or **until**—the epilogue code (if any) will be run, and any implicitly collected result will be returned as the value of the **loop**. For example,

```
(loop for x in '(1 2 3 4 5 6)
      collect x
      do (cond ((= x 4) (loop-finish))))
 => (1 2 3 4)
```

This particular example would be better written as **until (= x 4)** in place of the **do** clause.

19.2.7 Aggregated Boolean Tests

All of these clauses perform some test and may immediately terminate the iteration depending on the result of that test.

always *expr*
: Causes the loop to return **t** if *expr* always evaluates non-null. If *expr* evaluates to **nil** the loop immediately returns **nil**, without running the epilogue code (if any, as specified with the **finally** clause); otherwise, **t** will be returned when the loop finishes, after the epilogue code has been run.

never *expr*
: Causes the loop to return **t** if *expr* never evaluates non-null. This is equivalent to **always (not** *expr*).

thereis *expr*

If *expr* evaluates non-nil, then the iteration is terminated, and that value is returned without running the epilogue code.

19.2.8 Conditionalization

These clauses may be used to "conditionalize" the following clause. They may precede any of the side-effecting or value-producing clauses, such as do, collect, always, or return.

when *expr*
if *expr*

If *expr* evaluates to nil, the following clause will be skipped, otherwise not.

unless *expr*

This is equivalent to when (not *expr*)).

Multiple conditionalization clauses may appear in sequence. If one test fails, then any following tests in the immediate sequence, as well as the clause being conditionalized, are skipped.

Multiple clauses may be conditionalized under the same test by joining them with and, as in

```
(loop for i from a to b
      when (zerop (remainder i 3))
        collect i and do (print i))
```

which returns a list of all multiples of 3 from a to b (inclusive) and prints them as they are being collected.

If-then-else conditionals may be written using the else keyword, as in

```
(loop for i from a to b
      when (oddp i)
        collect i into odd-numbers
      else collect i into even-numbers)
```

Multiple clauses may appear in an else-phrase, using and to join them in the same way as above.

Conditionals may be nested. For example,

```
(loop for i from a to b
      when (zerop (remainder i 3))
        do (print i)
        and when (zerop (remainder i 2))
              collect i)
```

returns a list of all multiples of 6 from a to b, and prints all multiples of 3 from a to b.

When else is used with nested conditionals, the "dangling else" ambiguity is resolved by matching the else with the innermost when not already matched with an else. Here is a complicated example.

```
(loop for x in l
      when (atom x)
        when (memq x *distinguished-symbols*)
          do (process1 x)
        else do (process2 x)
      else when (memq (car x) *special-prefixes*)
             collect (process3 (car x) (cdr x))
             and do (memoize x)
           else do (process4 x))
```

Useful with the conditionalization clauses is the return clause, which causes an explicit return of its argument as the value of the iteration, bypassing any epilogue code. That is,

when *expr1* return *expr2*

is equivalent to

when *expr1* do (return *expr2*)

Conditionalization of one of the "aggregated boolean value" clauses simply causes the test that would cause the iteration to terminate early not to be performed unless the condition succeeds. For example,

```
(loop for x in l
      when (significant-p x)
        do (print x) (princ "is significant.")
        and thereis (extra-special-significant-p x))
```

does not make the extra-special-significant-p check unless the significant-p check succeeds.

The format of a conditionalized clause is typically something like

when *expr1 keyword expr2*

If *expr2* is the keyword it, then a variable is generated to hold the value of *expr1*, and that variable gets substituted for *expr2*. Thus, the composition

when *expr* return it

is equivalent to the clause

thereis *expr*

and one may collect all non-null values in an iteration by saying

when *expression* collect it

If multiple clauses are joined with and, the it keyword may only be used in the first. If multiple whens, unlesses, and/or ifs occur in sequence, the value substituted for it will be that of the last test performed. The it keyword is not recognized in an else-phrase.

19.2.9 Miscellaneous Other Clauses

named *name*

Defines a block named *name* around the code for the loop, so that one may use return-from to return explicitly out of this particular loop. This is obsolete now that block exists; it is cleaner to write (block *name* ...) around the loop.

Note that every loop generates a block named nil, so the function return can always be used to exit the innermost loop (assuming no other construct generating a block nil intervenes).

return *expression*

Immediately returns the value of *expression* as the value of the loop, without running the epilogue code. This is most useful with some sort of conditionalization, as discussed in the previous section. Unlike most of the other clauses, return is not considered to "generate body code", so it is allowed to occur between iteration clauses, as in

```
(loop for entry in list
      when (not (numberp entry))
        return (ferror ...)
      as frob = (times entry 2)
      ...)
```

Although ferror is called only for effect, return is used so that it can be called from that point in the loop.

If one instead desires the loop to have some return value when it finishes normally, one may place a call to the return function in the epilogue (with the finally clause, page 355).

19.3 Loop Synonyms

define-loop-macro *keyword* *Macro*

May be used to make *keyword*, a loop keyword (such as for), into a Lisp macro that may introduce a loop form. For example, after evaluating

```
(define-loop-macro for),
```

one may now write an iteration as

```
(for i from 1 below n do ...)
```

This facility exists primarily for diehard users of a predecessor of loop. Its unconstrained use is not recommended, as it tends to decrease the transportability of the code and needlessly uses up a function name.

19.4 Destructuring

Destructuring provides one with the ability to "simultaneously" assign or bind multiple variables to components of some data structure. Typically this is used with list structure. For example,

```
(loop with (foo . bar) = '(a b c) ...)
```

has the effect of binding foo to a and bar to (b c).

loop's destructuring support is intended to parallel and perhaps augment that provided by the host Lisp implementation, with a goal of minimally providing destructuring over list structure patterns. Thus, in Lisp implementations with no system destructuring support at all, one may still use list-structure patterns as loop iteration variables and in with bindings.

One may specify the data types of the components of a pattern by using a corresponding pattern of the data type keywords in place of a single data type keyword. This syntax remains unambiguous because wherever a data type keyword is possible, a loop keyword is the only other

possibility. Thus, if one wants to do

```
(loop for x in l
      as i = (car x)
      and j = (cadr x)
      and k = (cddr x)
      ...)
```

and no reference to x is needed, one may instead write

```
(loop for (i j . k) in l ...)
```

To allow some abbreviation of the data type pattern, an atomic component of the data type pattern is considered to state that all components of the corresponding part of the variable pattern are of that type. That is, the previous form could be written as

```
(loop for (i j . k) in l ...)
```

```
(defun map-over-properties (fn symbol)
   (loop for (propname propval) on (plist symbol) by 'cddr
         do (funcall fn symbol propname propval)))
```

maps *fn* over the properties on *symbol*, giving it arguments of the symbol, the property name, and the value of that property.

19.5 The Iteration Framework

This section describes the way loop constructs iterations. It is necessary if you will be writing your own iteration paths, and may be useful in clarifying what loop does with its input.

loop considers the act of *stepping* to have four possible parts. Each iteration-driving clause has some or all of these four parts, which are executed in this order:

pre-step-endtest
: This is an endtest which determines if it is safe to step to the next value of the iteration variable.

steps
: Variables that get stepped. This is internally manipulated as a list of the form (*var1 val1 var2 val2* ...); all of those variables are stepped in parallel, meaning that all of the *val*s are evaluated before any of the *var*s are set.

post-step-endtest
: Sometimes you can't see if you are done until you step to the next value; that is, the endtest is a function of the stepped-to value.

pseudo-steps
: Other things that need to be stepped. This is typically used for internal variables that are more conveniently stepped here, or to set up iteration variables that are functions of some internal variable(s) actually driving the iteration. This is a list like *steps*, but the variables in it do not get stepped in parallel.

The above alone is actually insufficient in just about all the iteration-driving clauses that loop handles. What is missing is that in most cases the stepping and testing for the first time through the loop is different from that of all other times. So, what loop deals with is two four-tuples as above; one for the first iteration, and one for the rest. The first may be thought of as describing code that immediately precedes the loop in the prog, and the second following the body code—in

fact, loop does just this, but severely perturbs it in order to reduce code duplication. Two lists of forms are constructed in parallel: one is the first-iteration endtests and steps, the other the remaining-iterations endtests and steps. These lists have dummy entries in them so that identical expressions will appear in the same position in both. When loop is done parsing all of the clauses, these lists get merged back together such that corresponding identical expressions in both lists are not duplicated unless they are "simple" and it is worth doing.

Thus, one *may* get some duplicated code if one has multiple iterations. Alternatively, loop may decide to use and test a flag variable that indicates whether one iteration has been performed. In general, sequential iterations have less overhead than parallel iterations, both from the inherent overhead of stepping multiple variables in parallel, and from the standpoint of potential code duplication.

One other point that must be noted about parallel stepping is that although the user iteration variables are guaranteed to be stepped in parallel, the placement of the endtest for any particular iteration may be either before or after the stepping. A notable case of this is

```
(loop for i from 1 to 3 and dummy = (print 'foo)
      collect i)
  => (1 2 3)
```

but prints foo *four* times. Certain other constructs, such as for *var* on, may or may not do this depending on the particular construction.

This problem also means that it may not be safe to examine an iteration variable in the epilogue of the loop form. As a general rule, if an iteration-driving clause implicitly supplies an endtest, then one cannot know the state of the iteration variable when the loop terminates. Although one can guess on the basis of whether the iteration variable itself holds the data upon which the endtest is based, that guess *may* be wrong. Thus,

```
(loop for subl on expr
      ...
      finally (f subl))
```

is incorrect, but

```
(loop as frob = expr while (g frob)
      ...
      finally (f frob))
```

is safe because the endtest is explicitly dissociated from the stepping.

19.6 Iteration Paths

Iteration paths provide a mechanism for user extension of iteration-driving clauses. The interface is constrained so that the definition of a path need not depend on much of the internals of loop. The typical form of an iteration path is

```
for var being {each|the} path {prepositionl exprl}...
```

path is an atomic symbol which is defined as a loop path function.
Any number of preposition/expression pairs may be present; the prepositions allowable for any particular path are defined by that path. For example,

```
(loop for x being the array-elements of my-array from 1 to 10
      ...)
```

To enhance readability, paths are usually defined in both the singular and plural forms; this

particular example could have been written as

```
(loop for x being each array-element of my-array from 1 to 10
      ...)
```

Another format, which is not so generally applicable, is

`for` *var* `being` *expr0* `and its` *path* {*preposition1 expr1*}...

In this format, *var* takes on the value of *expr0* the first time through the loop. Support for this format is usually limited to paths for which the next value is obtained by operating on the previous value. Thus, we can hypothesize the cdrs path, such that

```
(loop for x being the cdrs of '(a b c . d) collect x)
 => ((b c . d) (c . d) d)
```

but

```
(loop for x being '(a b c . d) and its cdrs collect x)
 => ((a b c . d) (b c . d) (c . d) d)
```

To satisfy the anthropomorphic among you, his, her, or their may be substituted for the its keyword, as may each. Egocentricity is not condoned. Some example uses of iteration paths are shown in section 19.6.1.

Very often, iteration paths step internal variables which the user does not specify, such as an index into some data-structure. Although in most cases the user does not wish to be concerned with such low-level matters, it is occasionally useful to have a handle on such things. loop provides an additional syntax with which one may provide a variable name to be used as an "internal" variable by an iteration path, with the using "prepositional phrase". The using phrase is placed with the other phrases associated with the path, and contains any number of keyword/variable-name pairs:

```
(loop for x being the array-elements of a using (index i)
      ...)
```

which says that the variable i should be used to hold the index of the array being stepped through. The particular keywords which may be used are defined by the iteration path; the index keyword is recognized by all loop sequence paths (section 19.6.1.3). Note that any individual using phrase applies to only one path; it is parsed along with the "prepositional phrases". It is an error if the path does not call for a variable using that keyword.

By special dispensation, if a *path* is not recognized, then the default-loop-path path will be invoked upon a syntactic transformation of the original input. Essentially, the loop fragment

`for` *var* `being` *frob*

is taken as if it were

`for` *var* `being default-loop-path in` *frob*

and

`for` *var* `being` *expr* `and its` *frob* ...

is taken as if it were

`for` *var* `being` *expr* `and its default-loop-path in` *frob*

Thus, this "undefined path hook" only works if the default-loop-path path is defined. Obviously, the use of this "hook" is competitive, since only one such hook may be in use, and the potential for syntactic ambiguity exists if *frob* is the name of a defined iteration path. This feature is not for casual use; it is intended for use by large systems that wish to use a special syntax for some feature they provide.

19.6.1 Pre-Defined Paths

loop comes with two pre-defined iteration path functions; one implements a mapatoms-like iteration path facility and the other is used for defining iteration paths for stepping through sequences.

19.6.1.1 The Interned-Symbols Path

The interned-symbols iteration path is like a mapatoms for loop.

```
(loop for sym being interned-symbols ...)
```

iterates over all of the symbols in the current package and its superiors.
This is the same set of symbols over which mapatoms iterates, although not necessarily in the same order. The particular package to look in may be specified as in

```
(loop for sym being the interned-symbols in package ...)
```

which is like giving a second argument to mapatoms.

You can restrict the iteration to the symbols directly present in the specified package, excluding inherited symbols, using the local-interned-symbols path:

```
(loop for sym being the local-interned-symbols  {in package}
      ...)
```

Example:

```
(defun my-apropos (sub-string &optional (pkg package))
    (loop for x being the interned-symbols in pkg
          when (string-search sub-string x)
            when (or (boundp x) (fboundp x) (plist x))
              do (print-interesting-info x)))
```

In the Zetalisp and NIL implementations of loop, a package specified with the in preposition may be anything acceptable to the pkg-find-package function. The code generated by this path will contain calls to internal loop functions, with the effect that it will be transparent to changes to the implementation of packages. In the Maclisp implementation, the obarray *must* be an array pointer, *not* a symbol with an array property.

19.6.1.2 The Hash-Elements Path

The hash-elements path provides an effect like that of the function maphash. It can find all the occupied entries in a hash table.

```
(loop for value being the hash-elements of hash-table ...)
```

iterates over all the occupied entries in *hash-table*. Each time, *value* is the value stored in the entry. To examine the keys of the entries as well, write

```
(loop for value being the hash-elements of hash-table
      with-key keysym ...)
```

and then keysym's value each will be the hash key that corresponds to *value*.

19.6.1.3 Sequence Iteration

One very common form of iteration is done over the elements of some object that is accessible by means of an integer index. loop defines an iteration path function for doing this in a general way and provides a simple interface to allow users to define iteration paths for various kinds of "indexable" data.

define-loop-sequence-path *Macro*

path-name-or-names fetch-fun size-fun &optional *sequence-type default-var-type*

path-name-or-names is either an atomic path name or list of path names. *fetch-fun* is a function of two arguments, the sequence and the index of the item to be fetched. (Indexing is assumed to be zero-origined.) *size-fun* is a function of one argument, the sequence; it should return the number of elements in the sequence. *sequence-type* is the name of the data-type of the sequence, and *default-var-type* the name of the data-type of the elements of the sequence. These are applicable to use of loop in other Lisp systems; on the Lisp Machine they might as well be omitted.

The Zetalisp implementation of loop utilizes the Zetalisp array manipulation primitives to define both array-element and array-elements as iteration paths:

```
(define-loop-sequence-path (array-element array-elements)
    aref array-active-length)
```

Then, the loop clause

```
for var being the array-elements of array
```

will step *var* over the elements of *array*, starting from element 0. The sequence path function also accepts in as a synonym for of.

The range and stepping of the iteration may be specified with the use of all of the same keywords which are accepted by the loop arithmetic stepper (for *var* from ...); they are by, to, downto, from, downfrom, below, and above, and are interpreted in the same manner. Thus,

```
(loop for var being the array-elements of array
          from 1 by 2
      ...)
```

steps *var* over all of the odd elements of *array*, and

```
(loop for var being the array-elements of array
          downto 0
      ...)
```

steps in reverse order.

```
(define-loop-sequence-path (vector-elements vector-element)
    vref vector-length notype notype)
```

is how the vector-elements iteration path can be defined in NIL (which it is). One can then do such things as

```
(defun cons-a-lot (item &restv other-items)
    (and other-items
         (loop for x being the vector-elements of other-items
               collect (cons item x))))
```

All such sequence iteration paths allow one to specify the variable to be used as the index variable, by use of the index keyword with the using prepositional phrase, as described (with an example) on page 364.

19.6.2 Defining Paths

This section and the next may not be of interest to those not interested in defining their own iteration paths.

In addition to the code which defines the iteration (section 19.5), a loop iteration clause (e.g. a for or as clause) produces variables to be bound and pre-iteration (*prologue*) code. This breakdown allows a user-interface to loop which does not have to depend on or know about the internals of loop. To complete this separation, the iteration path mechanism parses the clause before giving it to the user function that will return those items. A function to generate code for a path may be declared to loop with the define-loop-path function:

define-loop-path *Macro*

pathname-or-names path-function list-of-allowable-prepositions &rest *data*

This defines *path-function* to be the handler for the path(s) *path-or-names*, which may be either a symbol or a list of symbols. Such a handler should follow the conventions described below. The *datum-i* are optional; they are passed in to *path-function* as a list.

The handler will be called with the following arguments:

path-name

The name of the path that caused the path function to be invoked.

variable

The "iteration variable".

data-type

The data type supplied with the iteration variable, or nil if none was supplied. This is a facility of the loop intended for other Lisp systems in which declaring the type of a variable produces more efficient code. It is not documented in this manual since it is never useful on the Lisp Machine.

prepositional-phrases

This is a list with entries of the form *(preposition expression)*, in the order in which they were collected. This may also include some supplied implicitly (e.g. an of phrase when the iteration is inclusive, and an in phrase for the default-loop-path path); the ordering will show the order of evaluation that should be followed for the expressions.

inclusive?

This is t if *variable* should have the starting point of the path as its value on the first iteration (by virtue of being specified with syntax like for *var* being *expr* and its *path*), nil otherwise. When t, *expr* will appear in *prepositional-phrases* with the of preposition; for example, for x being foo and its cdrs gets *prepositional-phrases* of ((of foo)).

allowed-prepositions

This is the list of allowable prepositions declared for the path that caused the path

function to be invoked. It and *data* (immediately below) may be used by the path function such that a single function may handle similar paths.

data This is the list of "data" declared for the path that caused the path function to be invoked. It may, for instance, contain a canonicalized path, or a set of functions or flags to aid the path function in determining what to do. In this way, the same path function may be able to handle different paths.

The handler should return a list of either six or ten elements:

variable-bindings
This is a list of variables that need to be bound. The entries in it may be of the form *variable* or (*variable expression*).
Note that it is the responsibility of the handler to make sure the iteration variable gets bound. All of these variables will be bound in parallel; if initialization of one depends on others, it should be done with a **setq** in the *prologue-forms*. Returning only the variable without any initialization expression is not allowed if the variable is a destructuring pattern.

prologue-forms
This is a list of forms that should be included in the **loop** prologue.

the four items of the iteration specification
These are the four items described in section 19.5, page 362: *pre-step-endtest*, *steps*, *post-step-endtest*, and *pseudo-steps*.

another four items of iteration specification
If these four items are given, they apply to the first iteration, and the previous four apply to all succeeding iterations; otherwise, the previous four apply to *all* iterations.

Here are the routines that are used by **loop** to compare keywords for equality. In all cases, a *token* may be any Lisp object, but a *keyword* is expected to be an atomic symbol. In certain implementations these functions may be implemented as macros.

si:loop-tequal *token keyword*
This is the **loop** token comparison function. *token* is any Lisp object; *keyword* is the keyword it is to be compared against. It returns **t** if they represent the same token, comparing in a manner appropriate for the implementation.

si:loop-tmember *token keyword-list*
The **member** variant of **si:loop-tequal**.

si:loop-tassoc *token keyword-alist*
The **assoc** variant of **si:loop-tequal**.

If an iteration path function desires to make an internal variable accessible to the user, it should call the following function instead of **gensym**:

si:loop-named-variable *keyword*

This should only be called from within an iteration path function. If *keyword* has been specified in a **using** phrase for this path, the corresponding variable is returned; otherwise, **gensym** is called and that new symbol returned. Within a given path function, this routine should only be called once for any given keyword.

If the user specifies a **using** preposition containing any keywords for which the path function does not call **si:loop-named-variable**, **loop** will inform the user of his error.

19.6.2.1 Path Definition Example

Here is an example function that defines the **string-characters** iteration path. This path steps a variable through all of the characters of a string. It accepts the format

```
(loop for var being the string-characters of str ...)
```

The function is defined to handle the path by

```
(define-loop-path string-characters string-chars-path (of))
```

Here is the function:

```
(defun string-chars-path (path-name variable data-type
                          prep-phrases inclusive?
                          allowed-prepositions data
                          &aux (bindings nil)
                               (prologue nil)
                               (string-var (gensym))
                               (index-var (gensym))
                               (size-var (gensym)))
  allowed-prepositions data ; unused variables
  data-type
  ; To iterate over the characters of a string, we need
  ; to save the string, save the size of the string,
  ; step an index variable through that range, setting
  ; the user's variable to the character at that index.
  ; We support exactly one "preposition", which is required,
  ; so this check suffices:
  (cond ((null prep-phrases)
         (ferror nil "OF missing in ~S iteration path of ~S"
                 path-name variable)))
  ; We do not support "inclusive" iteration:
  (cond ((not (null inclusive?))
         (ferror nil
           "Inclusive stepping not supported in ~S path ~
            of ~S (prep phrases = ~:S)"
           path-name variable prep-phrases)))
  ; Set up the bindings
  (setq bindings (list (list variable nil)
                       (list string-var (cadar prep-phrases))
                       (list index-var 0)
                       (list size-var 0)))
  ; Now set the size variable
  (setq prologue (list '(setq ,size-var (string-length
                                          ,string-var))))
  ; and return the appropriate stuff, explained below.
  (list bindings prologue
        '(= ,index-var ,size-var)
        nil nil
        (list variable '(aref ,string-var ,index-var)
              index-var '(1+ ,index-var))))
```

The first element of the returned list is the bindings. The second is a list of forms to be placed in the *prologue*. The remaining elements specify how the iteration is to be performed. This example is a particularly simple case, for two reasons: the actual "variable of iteration", index-var, is purely internal (being gensymmed), and the stepping of it (1+) is such that it may be performed safely without an endtest. Thus index-var may be stepped immediately after

the setting of the user's variable, causing the iteration specification for the first iteration to be identical to the iteration specification for all remaining iterations. This is advantageous from the standpoint of the optimizations loop is able to perform, although it is frequently not possible due to the semantics of the iteration (e.g., for *var* first *expr1* then *expr2*) or to subtleties of the stepping. It is safe for this path to step the user's variable in the *pseudo-steps* (the fourth item of an iteration specification) rather than the "real" steps (the second), because the step value can have no dependencies on any other (user) iteration variables. Using the pseudo-steps generally results in some efficiency gains.

If one desired the index variable in the above definition to be user-accessible through the using phrase feature with the index keyword, the function would need to be changed in two ways. First, index-var should be bound to (si:loop-named-variable 'index) instead of (gensym). Secondly, the efficiency hack of stepping the index variable ahead of the iteration variable must not be done. This is effected by changing the last form to be

```
(list bindings prologue
      nil
      (list index-var '(1+ ,index-var))
      '(= ,index-var ,size-var)
      (list variable '(aref ,string-var ,index-var))
      nil
      nil
      '(= ,index-var ,size-var)
      (list variable '(aref ,string-var ,index-var)))
```

Note that although the second '(= ,index-var ,size-var) could have been placed earlier (where the second nil is), it is best for it to match up with the equivalent test in the first iteration specification grouping.