Tutorial for the Common Lisp Loop Macro

Peter D. Karp
SRI International
pkarp @ ai.sri.com

The Loop Macro is one of the most valuable, and least-well documented of the operations in Common Lisp. It is valuable because it is more powerful, more compact, and more readable than comparable Common Lisp constructs such as mapping operations and recursion. It also uses a programming style that will be familiar to programmers who have worked with other more traditional languages. This short guide provides examples of how to use the Loop Macro.

The Loop macro is different than most Lisp expressions in having a complex internal syntax that is more similar to programming languages like C or Pascal. So you need to read Loop expressions with half of your brain in Lisp mode, and the other half in Pascal mode.

Think of Loop expressions as having four parts: expressions that set up variables that will be iterated, expressions that conditionally terminate the iteration, expressions that do something on each iteration, and expressions that do something right before the Loop exits. In addition, Loop expressions can return a value. It is very rare to use all of these parts in a given Loop expression, but you can combine them in many ways.

;; Iterate through a list, and print each element.

```
(loop for x in '(a b c d e)
      do (print x) )

A
B
C
D
E
NIL
```

```
;; Iterate through two lists in parallel, and cons
;; up a result that is returned as a value by Loop.


(loop for x in '(a b c d e)
      for y in '(1 2 3 4 5)
      collect (list x y) )

((A 1) (B 2) (C 3) (D 4) (E 5))


;; Iterate using a counter, and a variable whose value
;; is computed from an expression on each iteration.

(loop for x from 1 to 5
      for y = (* x 2)
      collect y)

(2 4 6 8 10)


;; Iterate through a list, and have a counter iterate
;; in parallel.  The length of the list determines when
;; the iteration ends.  Two sets of actions are defined,
;; one of which is executed conditionally.
```

```
(loop for x in '(a b c d e)
      for y from 1

      when (> y 1)
      do (format t ", ")

      do (format t "~A" x)
      )

A, B, C, D, E
NIL


;; We could also write the preceding loop using the IF construct.

(loop for x in '(a b c d e)
      for y from 1

      if (> y 1)
      do (format t ", ~A" x)
      else do (format t "~A" x)
      )

A, B, C, D, E
NIL


;; Terminate the loop early using a test.  Actions can
;; consist of arbitrarily many lines and can refer to
;; variables defined outside the lexical scope of the loop.

  (loop for x in '(a b c d e 1 2 3 4)
        until (numberp x)
        do
        collect (list x 'foo))

((A FOO) (B FOO) (C FOO) (D FOO) (E FOO))


;; "While" can also serve as a termination check.  Both
;; "do" and "collect" can be combined in one expression.

(loop for x from 1
      for y = (* x 10)
      while (< y 100)

      do (print (* x 5))

      collect y)

5
10
15
20
25
30
35
40
45
(10 20 30 40 50 60 70 80 90)


;; Loops can be nested in various ways
```

```
(loop for x from 1 to 10
      collect (loop for y from 1 to x
                    collect y) )

((1) (1 2) (1 2 3) (1 2 3 4) (1 2 3 4 5) (1 2 3 4 5 6) (1 2 3 4 5 6 7)
 (1 2 3 4 5 6 7 8) (1 2 3 4 5 6 7 8 9) (1 2 3 4 5 6 7 8 9 10))


;; Several variables can loop through the components of a complex
;; list to destructure it.

(loop for (a b) in '((x 1) (y 2) (z 3))
      collect (list b a) )

((1 X) (2 Y) (3 Z))

;; Destructuring works for dotted pairs too.

(loop for (x . y) in '((1 . 1) (2 . 4) (3 . 9)) collect y)

(1 4 9)


;; The "return" action both stops the loop and returns a result.
;; Here we return the first numeric character in the string s.

(let ((s "alpha45"))
  (loop for i from 0 below (length s)
        for ch =  (char s i)
        when (find ch "0123456789" :test #'eql)
        return ch) )
#\4


;; Several actions provide shorthands for combinations of when/return

(loop for x in '(foo 2)
      thereis (numberp x))
T

(loop for x in '(foo 2)
      never (numberp x))
NIL

(loop for x in '(foo 2)
      always (numberp x))
NIL
```