LOOP
LOOP is a macro for expression iterations

1. (loop for item in '(a b c)
   for i from 0 by 2
   collect (list item i))

2. returns '((a 0) (b 2) (c 4))
Parts of a LOOP

- Step variables numerically & over data structures
- Collect, count, sum, minimize & maximize
- Execute Lisp code
- Decide when to terminate
- Conditionally do any of these
- Creating local variables for the LOOP
- Specify initial and final code
LOOP Keywords

- LOOP doesn’t use parantheses.

- LOOP keywords (for, from, by, collect) are recognized by their symbol-name!
Iteration Control: FOR

- Ranges of numbers, up or down, intervals
- Items / cons cells of a list
- Elements of vectors, incl. subtypes (=> strings)
- Entries in hash tables
- Symbols in a package
- Results of repeatedly evaluating forms
Counting Loops

- (loop for i): counts from 0 upwards
- (loop for i from 5)
- (loop for i to 10)
- (loop for i below 10)
- (loop for i from 5 below 15)
Counting Loops

• (loop for i from 0 downto -10)

• (loop for i from 0 above -10)

• (loop repeat 100)
LOOP over collections

- (loop for var in list)
- (loop for var on list)
- (loop for var across vector)
LOOP over collections

• (loop for key being the hash-keys in table)

• (loop for key being the hash-keys in table using (hash-value value))

• (loop for value being the hash-values in table)

• (loop for value being the hash-values in table using (hash-key key))
LOOP over collections

• (loop for sym being the symbols of package)

• (loop for sym being the present-symbols of ...)

• (loop for sym being the external-symbols of ...)

Equals-Then Iteration

- (loop repeat 5
  for x = 0 then y
  for y = 1 then (+ x y)
  collect y) => (1 2 4 8 16)

- (loop repeat 5
  for x = 0 then y
  and y = 1 then (+ x y)
  collect y) => (1 1 2 3 5)
Local Variables

• (loop with x = 10
   with y = (* x 2)
   ...
)

• (let* ((x 10)
         (y (* x 2)))
   (loop ...))
Local Variables

- (loop with \( x = 10 \)
  
  and \( y = (* \ x \ 2) \)
  
  ...)  

- (let ((\( x \) 10)
  
  (y (* \( x \) 2)))
  
  (loop ...)))
Destructuring

• (loop for (a b) in ‘((1 2) (3 4) (5 6))
  collect (* a b))
=> (2 12 30)

• (loop for (value . morep) on ‘(1 2 3 4 5) do
  (princ value)
  (when morep (princ "", "")))
=> 1, 2, 3, 4, 5
Destructuring

- (loop for (key value) on p-list by #'cddr
  unless (member key ‘(:accessor :reader :writer))
  append (list key value))

- (loop for (nil value) on p-list by #'cddr
  do (print value))
Value Accumulation

- (loop for (key value) on p-list by #'cddr
  collect key into keys
  collect value into values)

- (loop for (key value) on p-list by #'cddr
  unless (member key ‘(...))
  nconc (list key value) into new-p-list)
Value Accumulation

• (loop for i in numbers
  counting (evenp i) into evens
  counting (oddp i) into odds
  summing i into total
  maximizing i into max
  minimizing i into min)
Unconditional Execution

• (loop for i in list do (print i))

• (loop for i in list return i)
Conditional Execution

• (loop for i in list
   when (evenp i) sum i)

• (loop for i in list
   if (evenp i) maximize i into max-even
   else maximize i into max-odd)

• (loop for i in list
   unless (evenp i) sum i)
Initial / Final Execution

- (loop initially (print “I am beginning to loop.”))
  for i in list
  collect (* i 2)
  finally (print “I am ending the loop.”))

- finally clauses are not executed when the loop terminates early.
  (return, return-from)

- finally clauses see the variables of the loop.
Named LOOPS

- (loop named outer
  for list in lists do
    (loop for i in list
      when (got-it-p i) do
        (return-from outer i))
Termination Tests

• (loop for i in list
   while (< i 10)
   collect i)

• (loop for i in list
   until (< i 10)
   sum i)

• (loop for i in list do
   ... (loop-finish) ...)

Termination Tests

• (loop for i in list always (primep i))

• (loop for i in list thereis (oddp i))

• (loop for i in list never (oddp i))

• If those tests cause the loop to terminate, the finally clause is not executed!
Complete Loops

• First the named clause.

• Then initially, with, for and repeat.

• Then conditional, unconditional executions, accumulation and termination tests.

• Finally finally.