# Iteration/loops

- variety of iteration constructs provided with varying degrees of complexity, we'll only touch on a subset
- iteration inherently impure from a functional programming point of view
- could be implemented purely "under the hood" using tail recursive techniques
- will consider implementations and tradeoffs later

# dolist

- Simple construct for applying an action to each element of a list

```
(dolist  (varName theList) action_on_varName)
```

- e.g. assuming our list is L:

```
(dolist (e L) (format t “next element is ~A~%” e))
```

- It’s using e as a local variable name to refer to the current element during the iteration
- The “action” could of course be any valid lisp statement, e.g. a function call, a block, a let block, etc

# dotimes

- We specify a llocal counter variable, the number of times to repeat an action, what to return when finished, and the action to be performed

```
(dotimes (x N (foo x)) someAction)
```

- The local counter, x, goes from 0 to N-1, performing the action on each x value, then at the end it returns (foo x)

```
(dotimes (x 4 (* x x)) (format t "~A" x))
prints 0..3 then returns 16
```

# do

- The do construct is structured but powerful: we specify three key parts:
  - list of local variable definitions, each of which specifies the name, initial value, and how to update it each pass
  - list of the stopping condition then any actions to take and value to return once the stopping condition is met
  - anything left is the body of the loop (sequence of statements to execute each pass through the loop)

# do example

- Example: start with x=5, y=100, keep doubling x and incrementing y until x>y, and at each step we'll print both. When it stops, print x*y then return x+y.

```
(do ; first, a list of local variable settings
    ((x 5 (* x 2)) (y 100 (+ y 1)))
    ; second, a list of the stop condition and any actions,
    ; the last action determines the return value
    ((> x y) (format t "~A" (* x y)) (+ x y))
    ; and remaining statements are the body of the loop
    (format t "~A, ~A~%" x y))
```

# loop

- Loop: very flexible form, can specify a loop name, set of local variable specs, and a set of actions on the vars
- “for” is used within the loop to describe how a local variable is initialized/updated, e.g. for x in ‘(10 20 30 40), or for y upfrom 2 below 12 by 3 ; vals 2,5,8,11

```
(loop named Foo
   for x in ‘(10 20 30 40 50)
   for count from 5 ; increments count by 1 each time
   While (and (/= 30 x) (< count 7))
   do (...whatever with the current value of x,count))
```