

## Mapping functions over lists with `mapcar`

**Common programming task:** apply the same function to every element of a list, and return a list of results

`get-lengths` applies the `length` function to every list in an input list-of-lists:

```
(defun get-lengths (list1)
  (if (endp list1) ()
      (cons (length (first list1))
            (get-lengths (rest list1)))))

> (get-lengths '((a b) (have a nice day) (1 2 3)))
(2 4 3)
```

Fortunately, LISP provides a short-cut:

```
(defun get-lengths (list1)
  (mapcar #'length list1))
```



## More mapping with `mapcar`...

**Apply a predicate:**

```
> (mapcar #'zerop '(5 3 0 2 0 6))
(NIL NIL T NIL T NIL)
```

**Apply a function with multiple inputs:**

```
> (mapcar #'* '(1 2 3 4) '(6 2 1 4))
(6 4 3 16)

> (mapcar #'append '((one two) (three four))
           '((buckle my shoe) (shut the door)))
((ONE TWO BUCKLE MY SHOE) (THREE FOUR SHUT THE DOOR))
```

**Apply a user-defined function:**

```
> (defun add10 (num) (+ num 10))
> (mapcar #'add10 '(5 6 4 0 1))
(15 16 14 10 11)

> (defun add-not (list1) (cons 'do-not list1))
> (mapcar #'add-not '((fidget in class) (read e-mail in lab)
                    (copy homework)))
((DO-NOT FIDGET IN CLASS) (DO NOT READ E-MAIL IN LAB)
 (DO-NOT COPY HOMEWORK))
```

## Exercise:

```
> (make-opposites
    `(the enormous humorous elephant basked in the sun))
( (THE MINISCULE MELANCHOLY ELEPHANT BASKED IN THE MOON)

(defun make-opposites (list1)
  (mapcar
```

## Filtering lists with `remove-if`

**Another common programming task:** remove all elements from a list that satisfy a certain property

```
(defun remove-atoms (list1)
  (cond ((endp list1) ())
        ((atom (first list1)) (remove-atoms (rest list1)))
        (t (cons (first list1) (remove-atoms (rest list1))))))

> (remove-atoms '(a (b c) (d (e)) f (g) h))
((B C) (D (E)) (G))
```

**LISP again provides a short-cut:**

```
(defun remove-atoms (list1)
  (remove-if #'atom list1))

(defun bignum (num) (> num 100))

(remove-if #'bignum '(17 368 42 102 76 89 8576))
(17 42 76 89)
```

## Exercise:

```
> (remove-lists-with-duplicates
   '((a e f a) (g h k i q e s) (a b (a)) (j h f j w)))
((G H K I Q E S) (A B (A)))

(defun remove-lists-with-duplicates (list1)
  (remove-if
```

## Calling functions with `funcall`

The following two statements are equivalent:

```
(append '(gone with) '(the wind))
(funcall #'append '(gone with) '(the wind))
```

But why bother to invoke `funcall`?

Consider the search process:

```
(defun search (start goal method)
  ...
  ;; add new paths to queue
  ... (funcall method newpaths queue) ...
  ...)

(defun depth-first (new old) (append new old))
(defun breadth-first (new old) (append old new))

(search 'S 'G #'depth-first)
(search 'S 'G #'breadth-first)
```

## Defining local functions with `lambda`

`mapcar`, `remove-if` and `funcall` each have an input function that can be defined by the user ...

... but the functions we defined so far are very specialized  
e.g. `add10`, `add-not`, `bignum`

*Why clutter up the LISP environment with lots of little, specialized functions?*

**Instead, define a *local function* with `lambda`**

**function definition:** `(defun add10 (num) (+ num 10))`

**lambda expression:** `(lambda (num) (+ num 10))`

```
(mapcar #'(lambda (num) (+ num 10)) '(5 7 2 8))
(remove-if #'(lambda (num) (> num 100)) '(347 53 102 13))
(mapcar #'(lambda (phrase) (cons 'do-not phrase))
 '((fidget in class) (read e-mail in lab) (copy homework)))
```

## Exercise:

```
> (add-firsts-to-list
   '(do-not sometimes occasionally only-in-dire-emergency)
   '(read e-mail in lab))
(DO-NOT READ E-MAIL IN LAB) (SOMETIMES READ E-MAIL IN LAB)
(OCCASIONALLY READ E-MAIL IN LAB) (ONLY-IN-DIRE-EMERGENCY
READ E-MAIL IN LAB))
```

```
(defun add-firsts-to-list
  (mapcar #'(lambda
```

## Sorting things with sort

The `sort` function sorts the elements of a list using an arbitrary predicate to compare two elements of the list

```
> (sort '(6 9 2 6 1 7) #'<)
(1 2 6 6 7 9)
> (sort '(6 9 2 6 1 7) #'>)
(9 7 6 6 2 1)
```

**Beware!** `sort` destroys the list in the process!

```
> (setf numlist '(7 8 5 6 7 4))
(7 8 5 6 7 4)
> (sort numlist #'<)
(4 5 6 7 7 8)
> numlist
(7 7 8)
> (sort '((4 4 4 4) (1) (3 3 3) (2 2))
        #'(lambda (list1 list2)
            (< (length list1) (length list2))))
((1) (2 2) (3 3 3) (4 4 4 4))
```

## Exercise:

```
> (sort-by-cost '((7 (B A S)) (8 (D A S)) (4 (D S)))
                ((4 (D S)) (7 (B A S)) (8 (D A S))))
```

```
(defun sort-by-cost (list1)
  (sort list1
```

## Remember Java class definitions?

```
public class City
  private LinkedList neighbors;
  private LinkedList coords;
  public City (LinkedList neighbors, LinkedList coords) {
    this.neighbors = neighbors;
    this.coords = coords;
  }
  public LinkedList getNeighbors () {
    return neighbors;
  }
  public LinkedList getCoords () {
    return coords;
  }
  public void setNeighbors (LinkedList newNeighbors) {
    neighbors = newNeighbors;
  }
  public void setCoords (LinkedList newCoords) {
    coords = newCoords;
  }
}
```

## Creating object classes with defstruct

defstruct defines a new *structure*:

```
(defstruct city (neighbors nil) (coords nil))
```

Common LISP automatically provides constructor, reader and setter functions

```
(setf s (make-city :neighbors '(a d)
                  :coords '(-0.5 2.75))
(city-neighbors s)
(city-coords s)
(setf (city-neighbors s) '(a b d))
(setf (city-coords s) '(0.0 0.0))
```

## Formatted printing

format allows formatted printing of *strings*

```
> (format t "Hello!")
Hello!
NIL

(defun print-eec ()
  (format t "hist whist ~%little ghostthings")
  (format t "~%tip-toe ~%twinkle-toe"))

> (print-eec)
hist whist
little ghostthings
tip-toe
twinkle-toe
> (format t "The winner is ~a! Congrats!" 'Susie)
The winner is SUSIE! Congrats!
NIL
> (format t "Course: ~10a ~4a ~a" 'CS 232 'AI)
Course: CS          232 AI
NIL
```