Aside: mapcar

Lisp has a built-in function that is a more general version of our `consall`; it is called `mapcar`.

Example:

```
USER(3):
(mapcar 'length '( (1 2 3) (5) () (1 2 3 4) ))
(3 1 0 4)

; equivalent to:
USER(4): (consall '( (1 2 3) (5) () (1 2 3 4) ) 'le
(3 1 0 4)
```

`mapcar` can even apply a function that takes more than one argument; you just supply a list of values for the 1st argument, then a list of values for the second, etc.

Example:

```
USER(5): (mapcar '> '(99 2 5 -1) '(88 4 5 3))
(T NIL NIL NIL)
```


Example 2 (of passing functions): prune

Suppose we want a function that will test every element of a list and return a list containing only those that pass the test.

We want it to be very general: it should be able to use any test we might give it. How will we tell it what test to apply?

What should a function call look like?
Example: Prune out the elements of `myList` that are not atoms.

Now let's write the function.

```
; Return a new list containing only the elements of "list"
; that pass the test.
; Precondition:
```

```
(defun prune (list test)
    (cond ( (null list) nil )
          ( (funcall test (car list))
                 (cons (car list)
                       (prune (cdr list) test)
                 )
          )
          ( t (prune (cdr list) test) )
    )
)
```

Sample run

```
USER(12): (prune '((3 1) 4 (x y z) (x) y ()) 'atom)
(4 Y NIL)

USER(13): (prune '(() (a b c) (1 2) () (()) (x (y w) z)) 'nul
(NIL NIL)
```


Write calls to `prune` that will prune `myList` in these ways:

- Prune out elements that *are* null.
- (Assume `myList` contains lists of integers.) Prune out elements whose minimum is not at least 50.
  Hint: there is a built-in `min` function.
- (Assume `myList` contains lists.) Prune out elements that themselves have more than 2 elements.

This is becoming tedious. We need to declare a function for each possible test we might dream up.

Unnamed Functions

Fortunately, we can define functions without giving them a name. These are called “Lambda Expressions”.

Example definition:

```
(lambda (list) (< (length list) 2))
```

To call an unnamed function, just put the lambda-expression where you would normally put the function name. Example call:

```
( (lambda (list) (< (length list) 2))
  ‘( 1 2 3 4) )
```

Lambda expressions can have multiple parameters (like regular named functions) and any sort of function body that a named function could have.

Exercise: What is the value of each of these Lisp expressions?

```
( (lambda (x) (cons x nil)) ‘y )
```

```
( (lambda (x y) (> (length x) (length y)))
  ‘(a b c) ‘(d) )
```

```
( (lambda (x) (listp x)) ‘(lambda (x) (listp x)) )
```

```
( (lambda (x y) (append x y)) ‘(1 2) ‘(3 4 5) )
```

Using unnamed functions when calling prune

```
USER(39): (setq myList
               `(() (a b c) (1 2) () (()) (x (y w) z)))
(NIL (A B C) (1 2) NIL (NIL) (X (Y W) Z))

USER(40): (prune myList (lambda (x) (not (null x))))
((A B C) (1 2) (NIL) (X (Y W) Z))


USER(42): (setq myList `((59 72 40) (85 70 88 56)))
((59 72 40) (85 70 88 56))

USER(43): (prune myList (lambda (x) (> (apply `min x) 50)))
((85 70 88 56))



USER(52): (setq myList `(() (1) (1 2) (1 2 3) (1 2 3 4)))
(NIL (1) (1 2) (1 2 3) (1 2 3 4))

USER(53): (prune myList (lambda (x) (<= (length x) 2)))
(NIL (1) (1 2))
```


Why bother with lambda-expressions??

Example: Suppose we have tables of data (represented using Lisp lists), and functions that can do things like select out the rows of a given table that pass some test.

Suppose we want the *user* to be able to specify any criterion they might want. Examples:

- Retrieve students where gpa $>$ 3.0
- Retrieve courses where classSize $<$ 100
- Retrieve profs where building $=$ SF

It would be tedious to write a named function for every single criterion that the user might specify.

Instead, we can have the *program* construct an appropriate lambda-expression, based on the user's query.

Example 3 (of passing functions): bubblesort

```
; Precondition: smaller is a function that can be applied
; to two elements of the list.  It should return t iff the
; first argument is "smaller" than the second.

(defun bubblesort (list smaller)
    (helper list smaller (- (length list) 1))
)



; Bubblesorts the first n elements of list.  Returns a
; new list with the first n elements of "list" sorted,
; followed by the rest of "list" unchanged.
; Precondition: n < (length list).

(defun helper (list smaller n)

    (cond ( (eq n 0 ) list )
          ( t (helper (bubbleFirstN list smaller n)
                      smaller
                      (- n 1)
              )
          )
    )
)
```



```
; Does a single "bubble run".
; Precondition: n < (length list)

(defun bubbleFirstN (list smaller n)
    (cond ( (eq n 0) list )
          ( (funcall smaller (car list) (cadr list))
                (cons (car list)
                      (bubbleFirstN (cdr list)
                                    smaller
                                    (- n 1))
                )
          )
          ( t   (cons (cadr list)
                      (bubbleFirstN (cons (car list)
                                          (cddr list))
                                    smaller
                                    (- n 1))
                )
          )
    )
)
```

Is our `bubblesort` function $O(n^2)$, where n is the length of the original list, as it should be?

Can you improve the style and/or comments?

Sample run of function bubblesort

```
gardiner.cs 1% cl
Allegro CL 4.3 [SPARC; R1] (5/10/96 17:51) ...


USER(1): :ld sort.cl
; Loading ./sort.cl


USER(2): (bubblesort '(3 4 1 5 0 2 3) '<)
(0 1 2 3 3 4 5)


USER(3): (bubblesort
              '((a b c) (a) (1 2 3 4) () (z z z) (y y))
               (lambda (x y) (< (length x) (length y)))
         )
(NIL (A) (Y Y) (Z Z Z) (A B C) (1 2 3 4))


USER(4): (trace helper bubbleFirstN)
(BUBBLEFIRSTN HELPER)
```



```
USER(5): (bubblesort '(3 4 1 5 0 2 3) '<)
 0: (HELPER (3 4 1 5 0 2 3) < 6)
   1: (BUBBLEFIRSTN (3 4 1 5 0 2 3) < 6) ...
   1: returned (3 1 4 0 2 3 5)
   1: (HELPER (3 1 4 0 2 3 5) < 5)
     2: (BUBBLEFIRSTN (3 1 4 0 2 3 5) < 5) ...
     2: returned (1 3 0 2 3 4 5)
     2: (HELPER (1 3 0 2 3 4 5) < 4)
       3: (BUBBLEFIRSTN (1 3 0 2 3 4 5) < 4) ...
       3: returned (1 0 2 3 3 4 5)
       3: (HELPER (1 0 2 3 3 4 5) < 3)
         4: (BUBBLEFIRSTN (1 0 2 3 3 4 5) < 3) ...
         4: returned (0 1 2 3 3 4 5)
         4: (HELPER (0 1 2 3 3 4 5) < 2)
           5: (BUBBLEFIRSTN (0 1 2 3 3 4 5) < 2) ...
           5: returned (0 1 2 3 3 4 5)
           5: (HELPER (0 1 2 3 3 4 5) < 1)
             6: (BUBBLEFIRSTN (0 1 2 3 3 4 5) < 1) ...
             6: returned (0 1 2 3 3 4 5)
             6: (HELPER (0 1 2 3 3 4 5) < 0)
             6: returned (0 1 2 3 3 4 5)
           5: returned (0 1 2 3 3 4 5)
         4: returned (0 1 2 3 3 4 5)
       3: returned (0 1 2 3 3 4 5)
     2: returned (0 1 2 3 3 4 5)
   1: returned (0 1 2 3 3 4 5)
 0: returned (0 1 2 3 3 4 5)
(0 1 2 3 3 4 5)
USER(6): (exit)
; Exiting Lisp
gardiner.cs 2%
```

Eval

`eval` is a built-in function of one argument. It evaluates that argument twice and returns the result.

Example: Write an expression, to go inside function `blah`, that will return the sum of the elements of `list`.

```
(defun blah (list)




)
```


Defining an if-statement for Lisp

Define function `myif`, so that we can do things like this:

```
(myif (> num 0)      ; If num is greater than 0
      num            ; then return num
      (* num -1)     ; else return num * -1.
)
```

```
(defun myif (test then else)
    (cond (test then)
          (t    else)
    )
)
```

```
(defun absolute (num)
    (myif (> num 0)
          num
          (* num -1)
    )
)
```

```
(defun len (x)
    (myif (listp x)
          (length x)
          -1
    )
)
```


Problem: We shouldn't evaluate the arguments for `then` and `else` at the time of the call to `myif`.

Solution: Quote them.

```
(defun absolute (num)
    (myif (> num 0)
          'num
          '(* num -1)
    )
)
```

```
(defun len (x)
    (myif (listp x)
          '(length x)
          '-1
    )
)
```

Problem: The arguments for `then` and `else` are echoed back to us, unevaluated.

Solution: Eval them inside `myif`.

```
(defun myif (test then else)
    (cond (test (eval then))
          (t    (eval else))
    )
)
```


Problem: It may not make sense to evaluate the arguments for `then` and `else` in the context of `myif`.