

Using Functions That Take Functions as Arguments and Lambda Expressions to Define Your Own Functions

Here is a function, COUNT-ZEROS, that counts how many zeros appear in a list of numbers. It does this by taking the subset of the list elements that are zero, and then taking the length of the result.

```
(remove-if-not #'zerop '(34 0 0 95 0)) ⇒ (0 0 0)
```

From sec. 7.8 of
Touretzky's book.

```
(defun count-zeros (x)  
  (length (remove-if-not #'zerop x)))
```

```
(count-zeros '(34 0 0 95 0)) ⇒ 3
```

EXERCISES

7.11. Write a function to pick out those numbers in a list that are greater than one and less than five.

Note: To avoid using `x` with 2 meanings,
change `(lambda (x) (< 1 x 5))`
to `(lambda (y) (< 1 y 5))`.

Solution (on p. C-39):

```
7.11. (defun pick (x)  
  (remove-if-not #'(lambda (x) (< 1 x 5))  
    x))
```

- *A lambda expression anywhere in the body of a function f can use the formal parameters of f !*

Example from Touretzky:

6.6.4 SET-DIFFERENCE

The SET-DIFFERENCE function performs set subtraction. It returns what is left of the first set when the elements in the second set have been removed. Again, the order of elements in the result is undefined.

```
> (set-difference '(alpha bravo charlie delta)
                  '(bravo charlie))
(ALPHA DELTA)
```

```
> (set-difference '(alpha bravo charlie delta)
                  '(echo alpha foxtrot))
(BRAVO CHARLIE DELTA)
```

7.14. Here is a version of SET-DIFFERENCE written with REMOVE-IF:

```
(defun my-setdiff (x y)
  (remove-if #'(lambda (e) (member ? ?))
             x))
```

- *A lambda expression anywhere in the body of a function f can use the formal parameters of f !*

Example from Touretzky:

6.6.4 SET-DIFFERENCE

The SET-DIFFERENCE function performs set subtraction. It returns what is left of the first set when the elements in the second set have been removed. Again, the order of elements in the result is undefined.

```
> (set-difference '(alpha bravo charlie delta)
                  '(bravo charlie))
(ALPHA DELTA)
```

```
> (set-difference '(alpha bravo charlie delta)
                  '(echo alpha foxtrot))
(BRAVO CHARLIE DELTA)
```

7.14. Here is a version of SET-DIFFERENCE written with REMOVE-IF:

```
(defun my-setdiff (x y)
  (remove-if #'(lambda (e) (member e y))
            x))
```

Examples of the Use of MAPCAR, REMOVE-IF-NOT, and REMOVE-IF

```
(defun inc-list-2 (L n) ; cf. problem 3 of Assignment 4
  )
```

```
(defun neg-nums (L) ; cf. problem 2 of Assignment 4
  )
```

```
(defun neg-nums (L) ; cf. problem 2 of Assignment 4
  )
```

```
(defun partition (L p) ; cf. problem 7 of Assignment 4
  )
```

```
(defun singletons (L) ; cf. problem 13 of Assignment 4
  )
```

Examples of the Use of MAPCAR, REMOVE-IF-NOT, and REMOVE-IF

```
(defun inc-list-2 (L n) ; cf. problem 3 of Assignment 4
  (mapcar (lambda (x) (+ x n)) L))
```

```
(defun neg-nums (L) ; cf. problem 2 of Assignment 4
  (remove-if-not #'minusp L)) ; use built-in function
```

```
(defun neg-nums (L) ; cf. problem 2 of Assignment 4
  (remove-if-not (lambda (x) (< x 0)) L)); use lambda expr
```

```
(defun partition (L p) ; cf. problem 7 of Assignment 4
  (list (remove-if-not (lambda (x) (< x p)) L)
        (remove-if (lambda (x) (< x p)) L)))
```

```
(defun singletons (L) ; cf. problem 13 of Assignment 4
  (remove-if (lambda (x) (member x (cdr (member x L))))
            L))
```

Having seen examples of how to *use* functions that take functions as arguments, we now consider:

Question 2: How do we *write* functions that take functions as arguments?

Question 2: How do we *write* functions that take functions as arguments?

In Common Lisp the *value* of an identifier F (as a *variable*) and the *function definition* of F are two *unrelated and independent* attributes of F!

At any given time, an identifier F may have *neither*, *just one*, or *both* of these attributes.

In this regard Common Lisp is like Java (in which a class can have an instance variable named f and a method that's also named f) and unlike C++.

(defun f ...) sets the *function definition* of F, but doesn't affect the value (if any) of F.

The following will set the *value* of F but won't affect the function definition (if any) of F:

- (setf F ...)
- (let ((F ...)) or (let* ((F ...))
- Parameter passing (if F is a formal parameter).

Welcome to GNU CLISP 2.49 (2010-07-07) <<http://clisp.cons.org/>>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 100000))

F

[2]> (setf f 111)

111

[3]> (f f)

100111

[4]> (let ((f 222))
 (f f))

100222

[5]> (defun g (f) (f f))

G

[6]> (g 333)

100333

[7]>

Q: Can the value of a Lisp expression be a function?

A: Yes! Three important cases of this are:

- If G is a symbol that has a function definition, the value of `#'G` is G's function definition.
- The value of a lambda expression is a function.
- You can make the value of a variable G a function using SETF, LET / LET*, or parameter passing.

Q: When the value of a variable or other Lisp expression is a function, how can we *call* that function?

A: Use **FUNCALL** (or **APPLY**, which we'll look at later).

```
> (setf g #'(lambda (x) (* x 10)))  
#<Lexical-closure 41653824>
```

```
> (funcall g 12)  
120
```

From sec. 7.12 of
Touretzky.

The value of the variable G is a lexical closure, which is a function. But G itself is not the name of any function; if we wrote (G 12) we would get an undefined function error.

Welcome to GNU CLISP 2.49 (2010-07-07) <<http://clisp.cons.org/>>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993

Copyright (c) Bruno Haible, Marcus Daniels 1994-1997

Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998

Copyright (c) Bruno Haible, Sam Steingold 1999-2000

Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (defun f (x) (+ x 10000))

F

[2]> (defun g (x) (+ x 500))

G

[3]> (setf g #'f)

#<FUNCTION F ...

Value of G is the function named by F.

[4]> (g 1)

501

(g 1) calls the function named by G.

[5]> (funcall g 1)

10001

(funcall g 1) calls the function

[6]> (funcall #'g 1)

501

given by the value of G.

[7]>

(funcall #'g 1) calls the function

given by the value of #'G, which

is the function named by G.

FUNCALL Can Call Functions That Take 2 or More Args!

`(funcall f e_1 ... e_k)` calls the function given by the value of f ; the values of e_1, \dots, e_k are passed as arguments.

```
> (setf fn #'cons)  
#<Compiled-function CONS {6041410}>
```

```
> fn  
#<Compiled-function CONS {6041410}>
```

```
> (type-of fn)  
COMPILED-FUNCTION
```

```
> (funcall fn 'c 'd)  
(C . D)
```

This example is from sec. 7.2 of Touretzky.

It also serves as a reminder that if the 2nd arg of CONS isn't a list, CONS returns a dotted list!

The value of the variable FN is a function object. TYPE-OF shows that the object is of type COMPILED-FUNCTION. So you see that functions and symbols are not the same. The symbol CONS serves as the name of the CONS function, but it is not the actual function. The relationship between functions and the symbols that name them is explained in Advanced Topics section 3.18.

Writing Functions That Take Functions as Arguments

- When computing a call of a function `g`, each formal parameter of `g` is a variable whose value is set to the corresponding actual argument.
- So if a formal parameter `p` corresponds to an actual argument that is a function, then we must use `(funcall p ...)` to call that function.

`(p ...)` and `(funcall #'p ...)` would not work here: They'd call *the function whose name is p*, rather than the function given by parameter `p`'s value!

Example:

```
[1]> (defun yes-or-no (p x y)
      (if (funcall p x y)
          'yes
          'no))
YES-OR-NO
[2]> (yes-or-no #'> 30 29)
YES
[3]> (yes-or-no #'< 30 29)
NO
```

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
        an expression that computes (our-mapcar f L)
        from X and, possibly, f and/or L.
      ))))
```

To write , consider a specific example:

Let $L \Rightarrow (9\ 4\ 1\ 9\ 0)$, $f \Rightarrow n \mapsto \sqrt{n}$. So $(\text{cdr } L) \Rightarrow (4\ 1\ 9\ 0)$.
Now $X \Rightarrow (2\ 1\ 3\ 0)$
and we want \Rightarrow $(3\ 2\ 1\ 3\ 0)$
 $= (\text{cons } (\text{funcall } f (\text{car } L))\ X)$

From this we see that
 $(\text{cons } (\text{funcall } f (\text{car } L))\ X)$ is a good expression.

Writing Our Own Version of MAPCAR

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
        (cons (funcall f (car L)) X) ))))
```

X isn't used more than once, so we eliminate the LET:

```
(defun our-mapcar (f L)
  (if (endp L)
      nil
      (let ((X (our-mapcar f (cdr L))))
        (cons (funcall f (car L)) (our-mapcar f (cdr L))) ⇒)))
```

Writing the SIGMA Function

Recall that SIGMA should behave as follows:

If $g \Rightarrow$ a numerical function of one argument
and $j, k \Rightarrow$ integers,
then $(\text{sigma } g \ j \ k) \Rightarrow g(j) + g(j+1) + \dots + g(k)$.
[This sum is 0 if $j > k$.]

Q. What should we make smaller for the recursive call?

A. The *no. of summands* ($k-j+1$ in the above example).

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        an expression that computes (sigma g j k)
        from X and, possibly, g, and/or j, and/or k
      )))
```

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        an expression that computes (sigma g j k)
        from X and, possibly, g, and/or j, and/or k
      )))
```

To write , consider a specific example:

Let $g \Rightarrow x \mapsto x^2$, $j \Rightarrow 3$, and $k \Rightarrow 6$. So $(+ j 1) \Rightarrow 4$.
Now $X \Rightarrow 4^2 + 5^2 + 6^2$
and we want $\Rightarrow 3^2 + 4^2 + 5^2 + 6^2$
 $= (+ (\text{funcall } g \ j) \ X)$

From this we see that
 $(+ (\text{funcall } g \ j) \ X)$ is a good expression.

Writing the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        (+ (funcall g j) X))))
```

X isn't used more than once, so we eliminate the LET:

```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        (+ (funcall g j) (sigma g (+ j 1) k)) ⇒))
```

Using the SIGMA Function

```
(defun sigma (g j k)
  (if (> j k)
      0
      (+ (funcall g j) (sigma g (+ j 1) k))))
```


The following function

sum-of-kth-powers-of-1st-n-positive-integers

does what its name suggests; for example:

```
(sum-of-kth-powers-of-1st-n-positive-integers 5 9)
=> 15 + 25 + 35 + 45 + 55 + 65 + 75 + 85 + 95 = 120825
```

```
(defun sum-of-kth-powers-of-1st-n-positive-integers (k n)
  (sigma (lambda (i) (expt i k)) 1 n))
```



expt is a built-in Common Lisp function that performs exponentiation: **(expt i k) => i^k**

- As we have seen, functions given by lambda expressions can be passed as arguments to other functions. They can also be returned to the caller of a function (as the result of the function call).
- Suppose a function f has a formal parameter k , and a lambda expression in the body of f uses k . **Then, regardless of where the function given by that lambda expression is called, the variable name k in the lambda expression's body will denote the formal parameter k of the call of f that created the lambda expression.**
 - The function given by a Common Lisp lambda expression is represented as a closure (also called a lexical closure): This representation provides access to f 's parameter k during execution of the function given by the lambda expression in the above example.

A Concrete Example:

Recall the following function from an earlier slide:

```
(defun sum-of-kth-powers-of-1st-n-positive-integers (k n)
  (sigma (lambda (i) (expt i k)) 1 n))
```

This function passes `(lambda (i) (expt i k))` into a call of the function SIGMA, which we defined as follows:

```
(defun sigma (g j k)
  (if (> j k)
      0
      (+ (funcall g j) (sigma g (+ j 1) k))))
```

When `(lambda (i) (expt i k))` is called inside SIGMA's body, `k` in the lambda expression still refers to the parameter `k` of `sum-of-kth-powers-of-1st-n-positive-integers`!

- Similarly, a *lambda expression anywhere in the body of a LET or LET* form can use the local variables of the LET or LET**: The closure representation of the function given by a lambda expression provides access to any such local variables that are used in the lambda expression when the function is executed.

Note: In addition to ordinary variables, Common Lisp also supports so-called "special" variables that are dynamically scoped, but special variables will **not** be used in this course. The above remarks assume that no formal parameter or local variable is a special variable.

In the above example, if *k* were a special variable of SIGMA and sum-of-kth-powers-of-1st-n-positive-integers, then *k* *in the lambda expression would refer to the parameter k of SIGMA and not the parameter k of SUM-OF-KTH-POWERS-OF-1ST-N-POSITIVE-INTEGERS!*

MAPCAR Can Also Map Functions of Two or More Arguments

7.11 OPERATING ON MULTIPLE LISTS

From Touretzky's book

In the beginning of this chapter we used MAPCAR to apply a one-input function to the elements of a list. MAPCAR is not restricted to one-input functions, however. Given a function of n inputs, MAPCAR will map it over n lists. For example, given a list of people and a list of jobs, we can use MAPCAR with a two-input function to pair each person with a job:

```
> (mapcar #'(lambda (x y) (list x 'gets y))
      '(fred wilma george diane)
      '(job1 job2 job3 job4))
((FRED GETS JOB1)
 (WILMA GETS JOB2)
 (GEORGE GETS JOB3)
 (DIANE GETS JOB4))
```

MAPCAR goes through the two lists in parallel, taking one element from each at each step. If one list is shorter than the other, MAPCAR stops when it reaches the end of the shortest list.

Another example of operating on multiple lists is the problem of adding two lists of numbers pairwise:

```
> (mapcar #'+ '(1 2 3 4 5) '(60 70 80 90 100))
(61 72 83 94 105)

> (mapcar #'+ '(1 2 3) '(10 20 30 40 50))
(11 22 33)
```

Three More Examples of MAPCAR

```
[1]> (mapcar #'(10000 20000 30000 40000)
      '(1000 2000 3000 4000)
      '(100 200 300 400)
      '(10 20 30 40)
      '(1 2 3 4))
(11111 22222 33333 44444)
[2]>
```

```
=====
[1]> (mapcar #'append '((A B) (1 2 3 4) (X))
                  '((P) (Q R) (S T U))
                  '((X X X X) (Y Y Y) (Z Z))
                  '((+ - *) (/ %) (^)))
((A B P X X X X + - *) (1 2 3 4 Q R Y Y Y / %) (X S T U Z Z ^))
[2]>
```

```
=====
[1]> (mapcar (lambda (x y z) (list '1st x '2nd y '3rd z))
            '(ORANGE PLUM APPLE PEAR)
            '(ALPHA BETA GAMMA DELTA)
            '(MOUSE CAT RAT DOG))
((1ST ORANGE 2ND ALPHA 3RD MOUSE) (1ST PLUM 2ND BETA 3RD CAT)
 (1ST APPLE 2ND GAMMA 3RD RAT) (1ST PEAR 2ND DELTA 3RD DOG))
[2]>
```