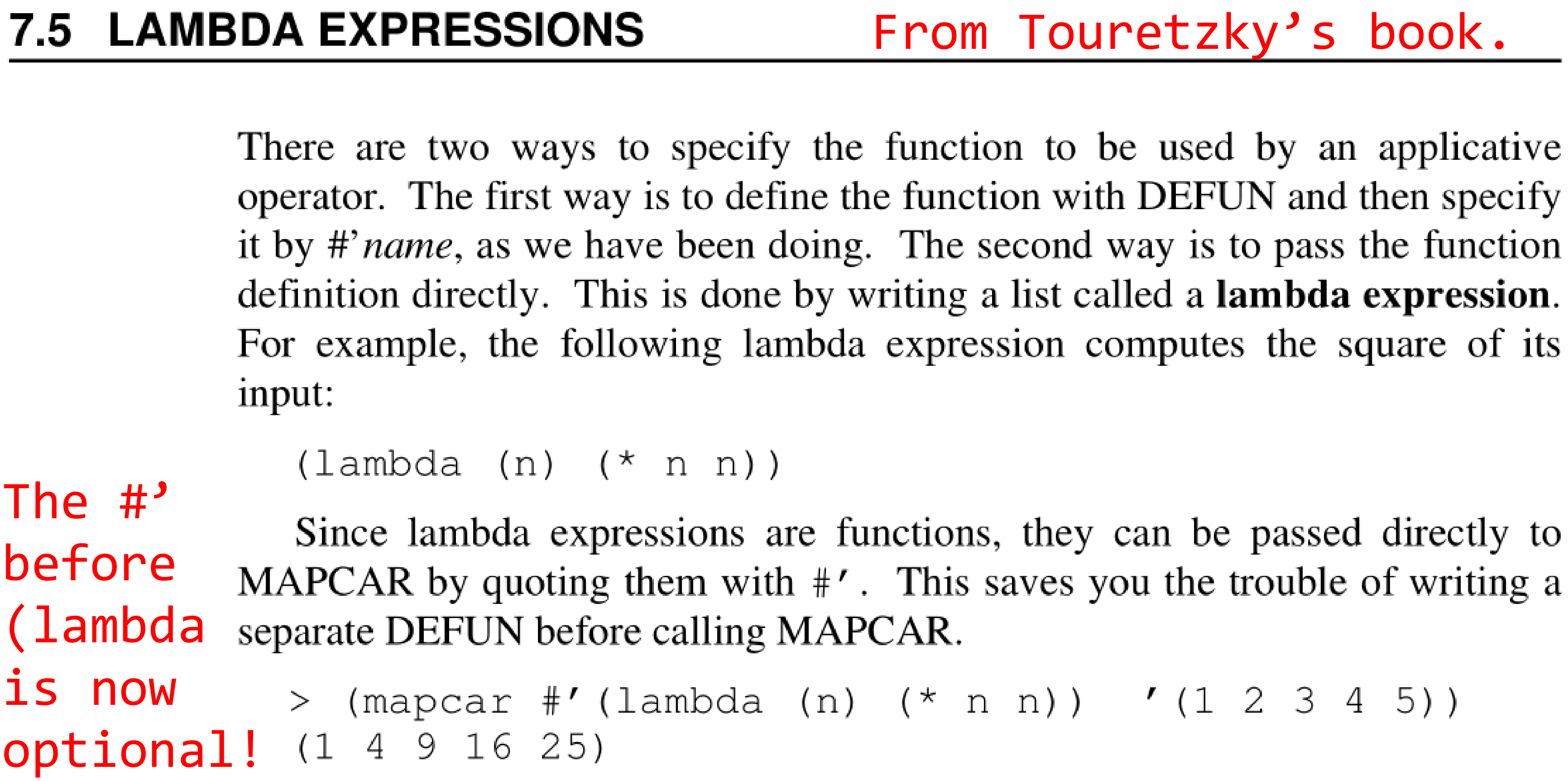
If we don’t expect to use the GREATER-THAN-FIVE-P function of exercise 7.4 elsewhere, we can give a more concise solution to the exercise: We can *use a* ***lambda expression*** to *create the function without naming it.*

From Touretzky’s book.



7.5 LAMBDA EXPRESSIONS

There are two ways to specify the function to be used by an applicative operator. The first way is to define the function with DEFUN and then specify it by #’*name*, as we have been doing. The second way is to pass the function definition directly. This is done by writing a list called a **lambda expression**. For example, the following lambda expression computes the square of its input:

```
(lambda (n) (* n n))
```

Since lambda expressions are functions, they can be passed directly to MAPCAR by quoting them with #′. This saves you the trouble of writing a separate DEFUN before calling MAPCAR.

```
> (mapcar #′(lambda (n) (* n n)) ′(1 2 3 4 5))
(1 4 9 16 25)
```

The #’ before (lambda is now optional!

7.8 REMOVE-IF AND REMOVE-IF-NOT

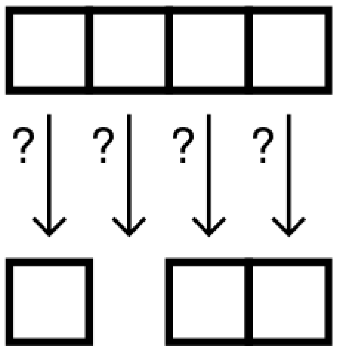
From Touretzky's book.

REMOVE-IF is another applicative operator that takes a predicate as input. REMOVE-IF removes all the items from a list that satisfy the predicate, and returns a list of what's left.

```
> (remove-if #'numberp '(2 for 1 sale))
(FOR SALE)

> (remove-if #'oddp '(1 2 3 4 5 6 7))
(2 4 6)
```

Here is a graphical description of REMOVE-IF:



The REMOVE-IF-NOT operator is used more frequently than REMOVE-IF. It works just like REMOVE-IF except it automatically inverts the sense of the predicate. This means the only items that will be removed are those for which the predicate returns NIL. So REMOVE-IF-NOT returns a list of all the items that *satisfy* the predicate. Thus, if we choose PLUSP as the predicate, REMOVE-IF-NOT will find all the positive numbers in a list.

```
> (remove-if-not #'plusp '(2 0 -4 6 -8 10))
(2 6 10)
   > (remove-if-not #'oddp '(2 0 -4 6 -8 10))
NIL
```

From Touretzky's Book

Here are some additional examples of REMOVE-IF-NOT:

```
> (remove-if-not #'(lambda (x) (> x 3))
                  '(2 4 6 8 4 2 1))
(4 6 8 4)

> (remove-if-not #'numberp
       '(3 apples 4 pears and 2 little plums))
(3 4 2)

> (remove-if-not #'symbolp
        '(3 apples 4 pears and 2 little plums))
(APPLES PEARS AND LITTLE PLUMS)
```

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)

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

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

From sec. 7.8 of Touretzky's book.

EXERCISES

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

Solution (on p. C-39):

Note: (lambda (x) (< 1 x 5)) could be changed to (lambda (y) (< 1 y 5))

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

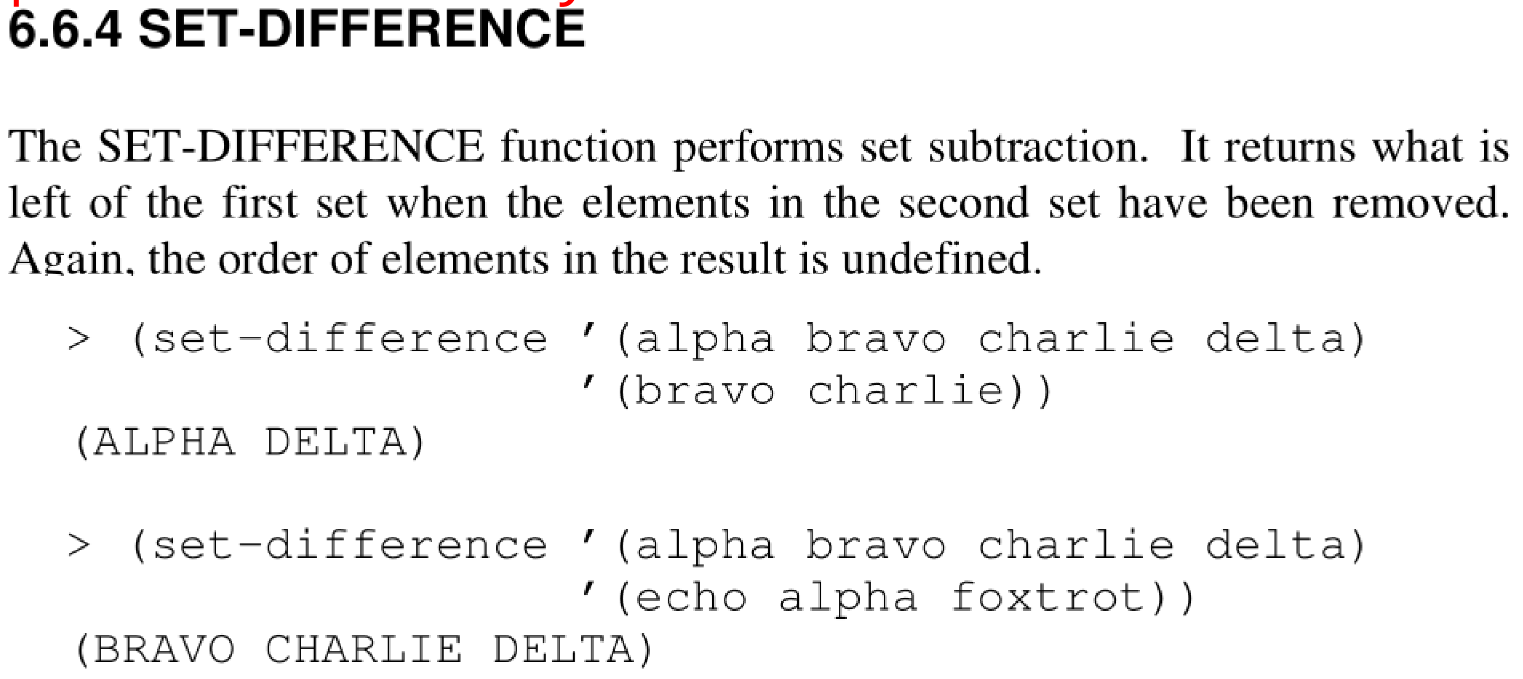
- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

- A ***closure*** is a function represented in the machine in such a way that, regardless of when and where the function is called, *the function has access to any variable that is in scope at the point where the function is defined or created.*

- When a function is passed as an argument into or returned as the result of a function call, the argument or result is a ***closure***.

- A lambda expression can use any variable that is in scope where the expression appears--e.g., *if it is in a function it can use the function's parameters*!

Example from Touretzky:

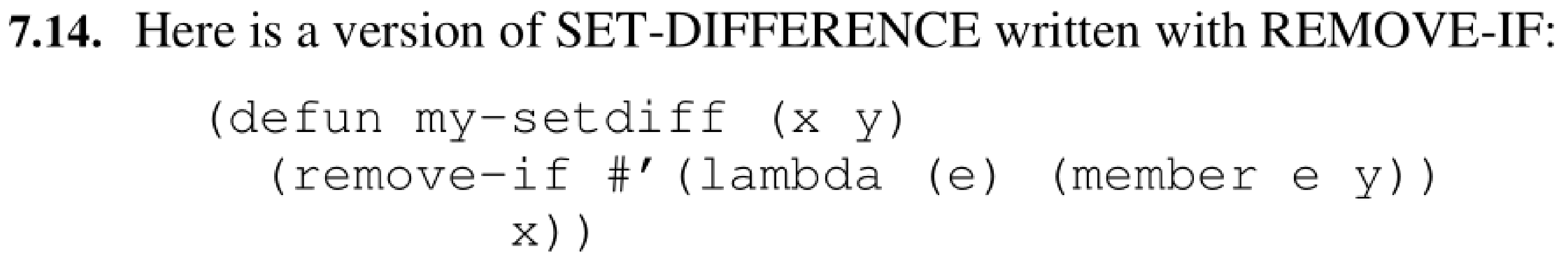


6.6.4 SET-DIFFERENCE

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))
```

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!

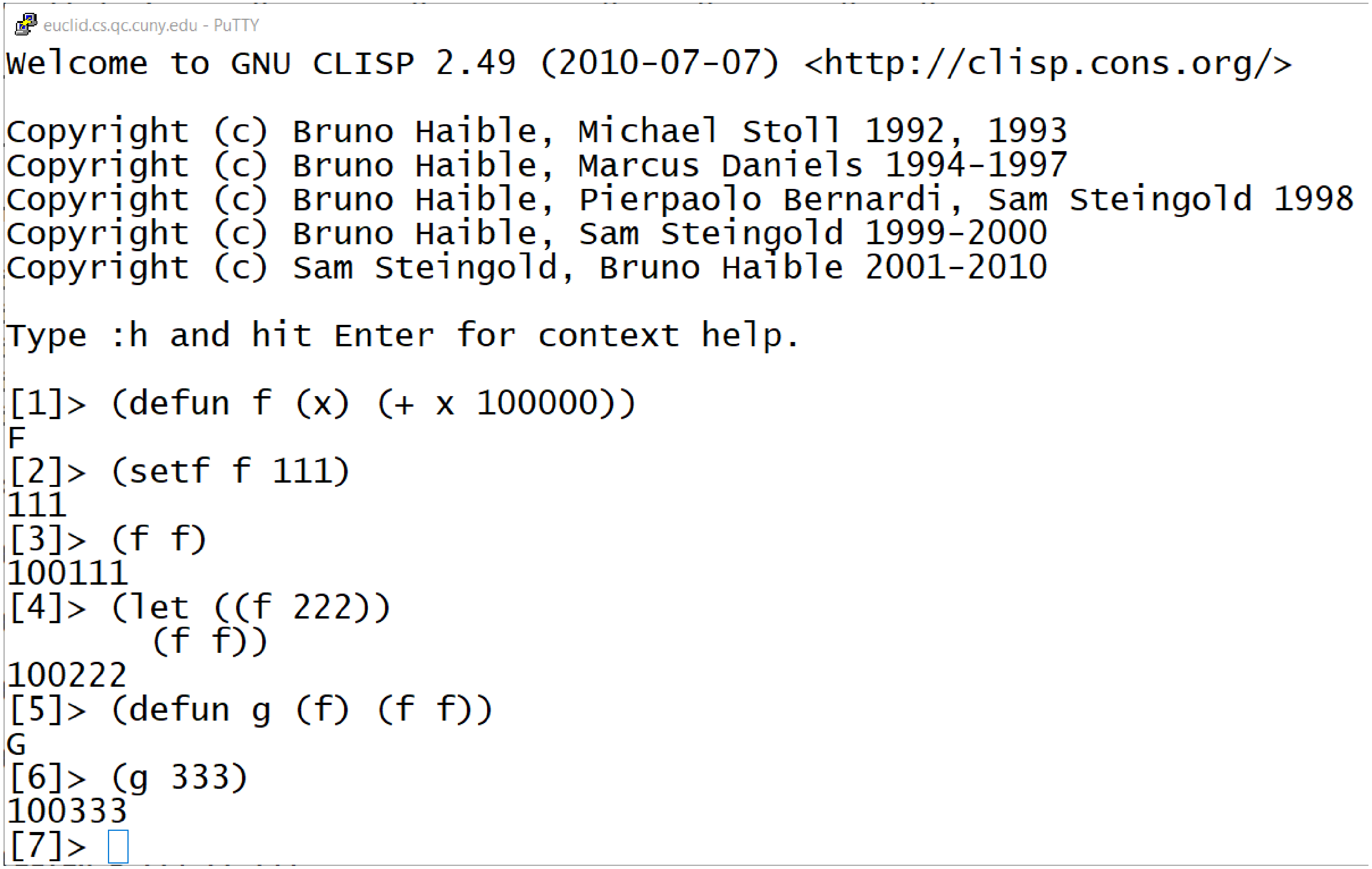
At any given time, an identifier F may have *neither*, *just one*, or *both* of these attributes.

In this regard Common Lisp is like Java (which allows, e.g., a class to have both a method F and a field 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]> 
```

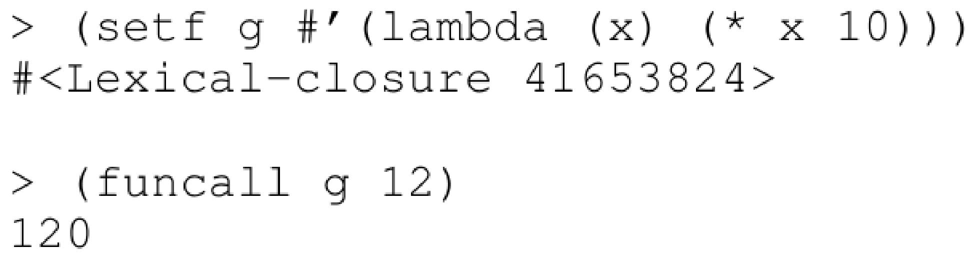
Q: Can the ***value*** of a Lisp expression be a function?

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?

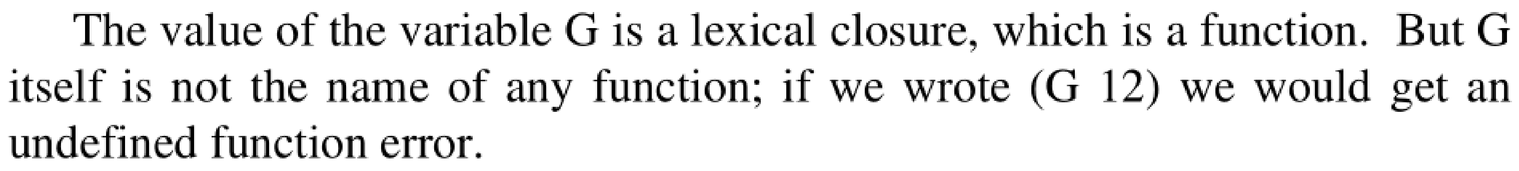
A: Use **FUNCALL** (or APPLY, which we’ll look at later).



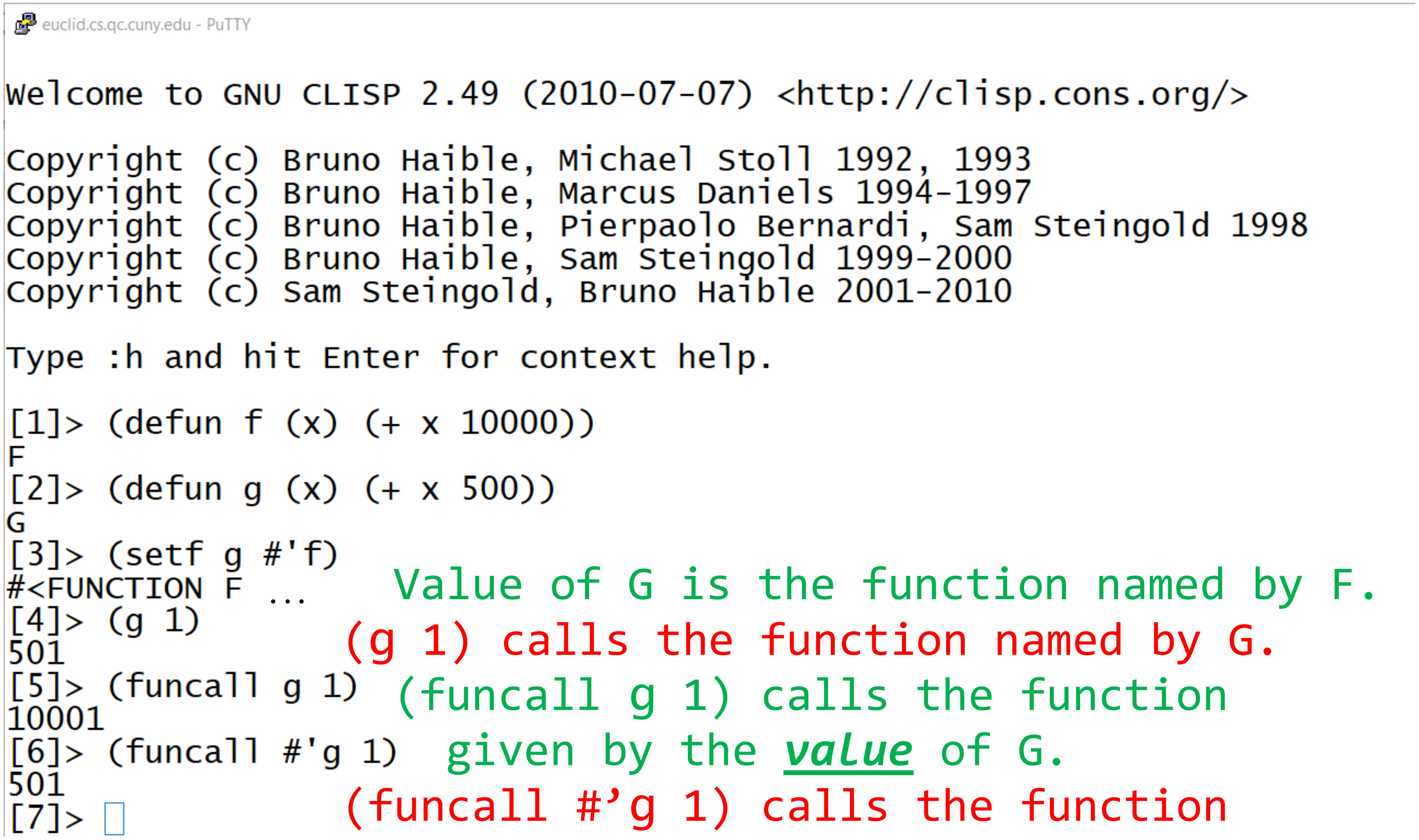
```
> (setf g #'(lambda (x) (* x 10)))
#<Lexical-closure 41653824>

> (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 ...
[4]> (g 1)
501
[5]> (funcall g 1)
10001
[6]> (funcall #'g 1)
501
[7]>
```

Value of G is the function named by F.

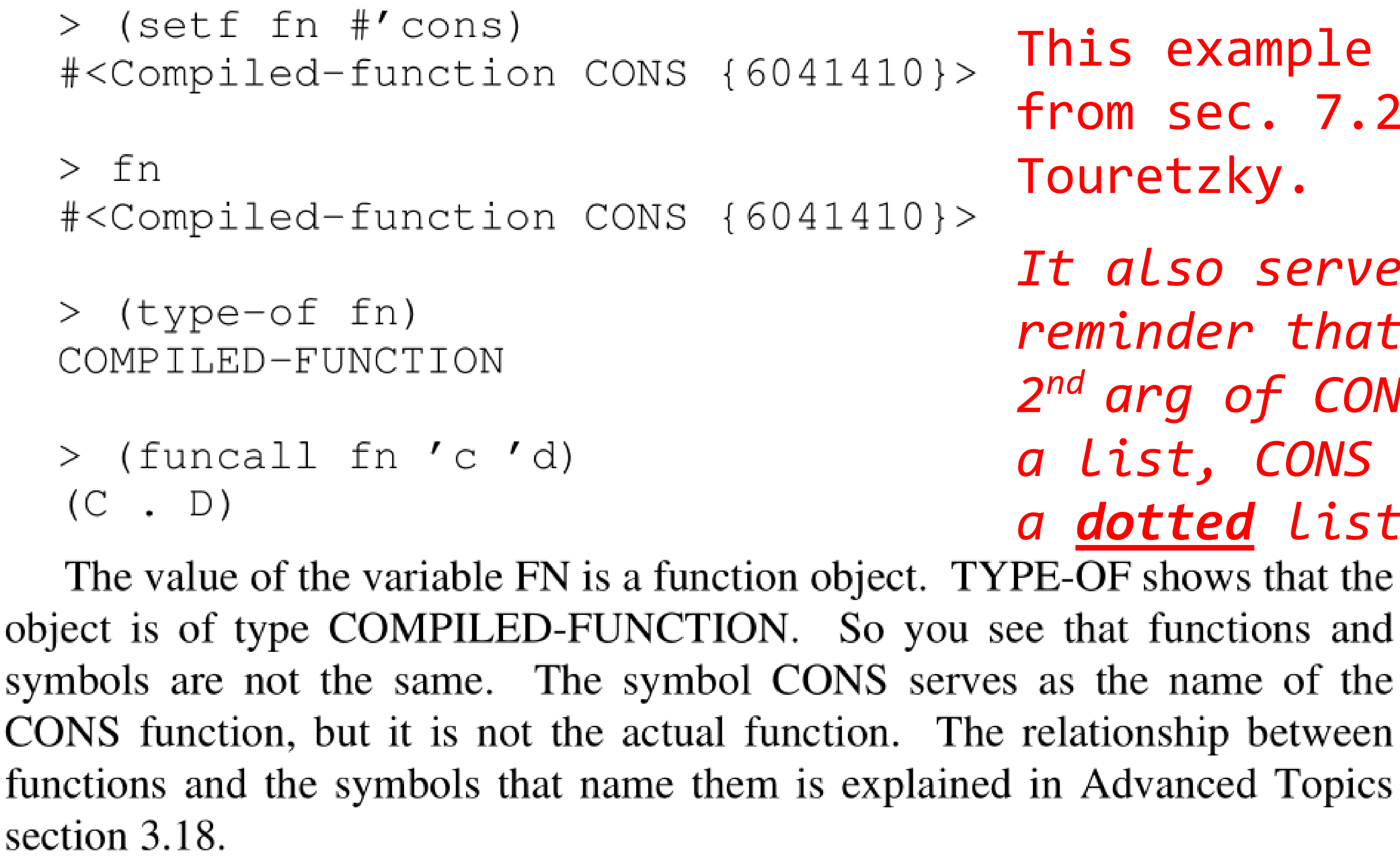
(g 1) calls the function named by G.

(funcall g 1) calls the function given by the ***value*** of G.

(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, ..., 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)
```

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.

This example is from sec. 7.2 of Touretzky.

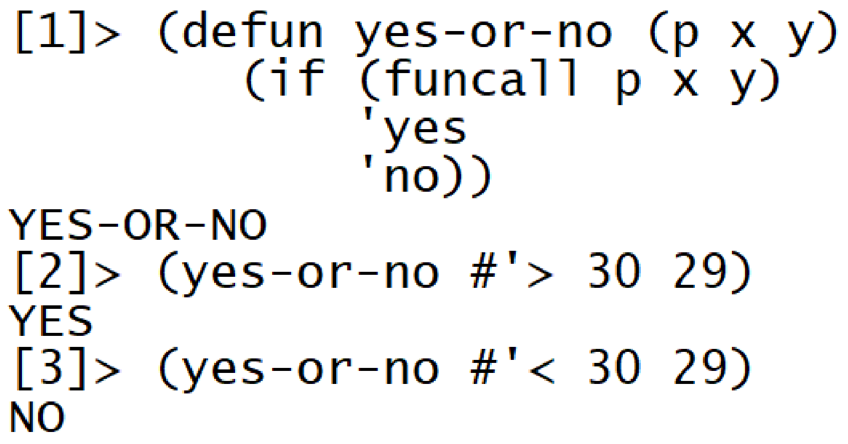
It also serves as a reminder that if the 2nd arg of CONS isn't a list, CONS returns a ***dotted*** *list!*

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 $\boxed{}$, suppose L, f => (9 4 1 16 0), $n \mapsto \sqrt{n}$.

```
Since   (MAPCAR #'sqrt '(9 4 1 16 0)) => (3 2 1 4 0)
and     (MAPCAR #'sqrt   '(4 1 16 0)) =>   (2 1 4 0)
we see that:
```

If L => (9 4 1 16 0) and f => $n \mapsto \sqrt{n}$

then X => (2 1 4 0)

and we want $\boxed{}$ **to** => (3 2 1 4 0)

= (cons (funcall f (car L)) X)

From this we see that

(cons (funcall f (car L)) X) is a good $\boxed{}$ 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 => a numerical function of one argument
   and  j, k => integers,
   then (sigma g j k) => g(j) + g(j+1) + ... + 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 $\boxed{}$, suppose g => $x \mapsto x^2$, j => 3, and k => 6.
As (SIGMA (lambda (x) (* x x)) 3 6) => $3^2 + 4^2 + 5^2 + 6^2$
and (SIGMA (lambda (x) (* x x)) 4 6) => $4^2 + 5^2 + 6^2$
we see that:

If g => $x \mapsto x^2$, j => 3, and k => 6
then X => $4^2 + 5^2 + 6^2$
and we want $\boxed{}$ **to** => $3^2 + 4^2 + 5^2 + 6^2$
= (+ (funcall g j) X)

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

Writing the SIGMA Function

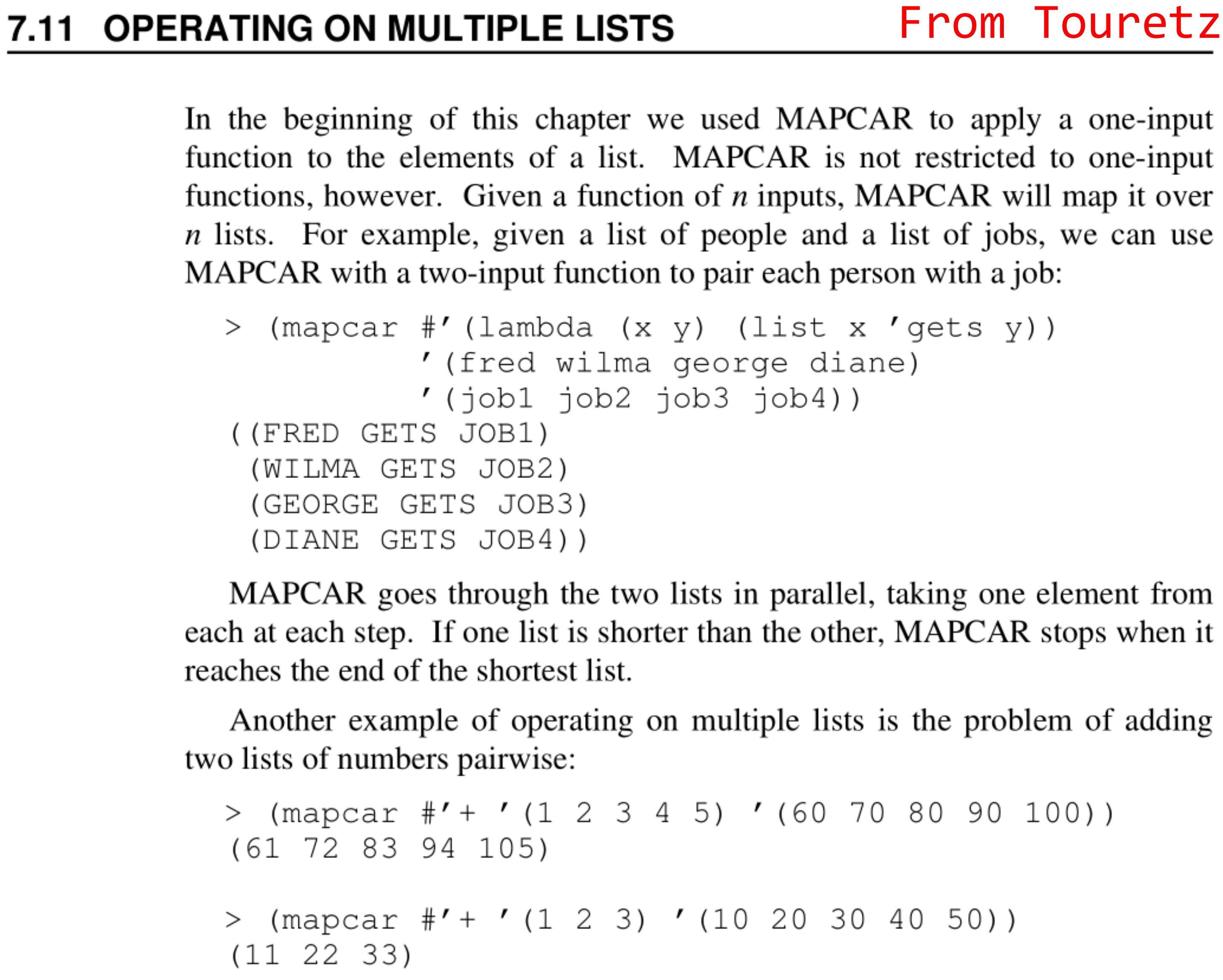
```
(defun sigma (g j k)
  (if (> j k)
      0
      (let ((X (sigma g (+ j 1) k)))
        (+ (funcall g j) X) )))
```

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)) ~~)~~))
```

MAPCAR Can Also Map Functions of Two or More Arguments

From Touretzky's book



7.11 OPERATING ON MULTIPLE LISTS

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)
```