Example of the Use of **(floor n 2)** as a Recursive Call Argument

e (i.e., the base of natural logs) is one of the best known constants. How can we calculate e very accurately? To be concrete, let's say we want to find a number y such that:

$$y < e < 1.000000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

One way is to use the following fact (which we'll assume is true but isn't very hard to prove if you know enough calculus):

$$\left(1+\frac{1}{n}\right)^{n} < e < \left(1+\frac{1}{n}\right)\left(1+\frac{1}{n}\right)^{n} = \left(1+\frac{1}{n}\right)^{n+1}$$

When $n = 10^{25}$, this fact says that $(\clubsuit)$ holds when y is

$$(1+10^{-25})^{10^{25}} = 1.000000000000000000000000001^{10000000000000000000000000}$$

Q. How can we write a recursive function **power** such that

(power z n) $\Rightarrow$ z^n if z $\Rightarrow$ a number & n $\Rightarrow$ an integer ≥ 0

that can be used to compute $(1+10^{-25})^{10^{25}}$?

Example of the Use of (floor n 2) as a Recursive Call Argument

e (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (♣)$$

It can be shown using calculus that (♣) holds when *y* is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

Q. How can we write a recursive function **power** such that

(power z n) ⇒ z^n if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1 + 10^{-25})^{10^{25}}$?

- We ***cannot*** use a definition based on $z^n = z * z^{n-1}$ such as

```
(defun power (z n)    ; far too inefficient!
  (cond ((= n 0) 1)
        (t (* z (power z (- n 1))))))
```

because when we pass 10^{25} to n this function would need a recursion depth of 10^{25}, which would ***require an impossibly large amount of memory***; and it'd also ***take an impossibly long time*** to execute 10^{25} calls of power!

Example of the Use of (floor n 2) as a Recursive Call Argument

We want to find a number y such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

It can be shown using calculus that ($\clubsuit$) holds when y is

$$(1+10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

Q. How can we write a recursive function **power** such that

(power z n) $\Rightarrow$ z^n if z $\Rightarrow$ a number & n $\Rightarrow$ an integer ≥ 0

that can be used to compute $(1+10^{-25})^{10^{25}}$?

- A solution is given by the function below:

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

- We get (floor n 2) by ***chopping off the rightmost bit of*** n.
- As $2^{83} < 10^{25} < 2^{84}$, the binary representation of 10^{25} has 84 bits: So a call of power with 10^{25} as the value of n makes a total of just 84 recursive calls!

This function **power** can now be used in Clisp to compute the number y that satisfies $y < e < (1 + 10^{-25})\,y$.

```
euclid> cl
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

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]> (load "power.lsp")
;; Loading file power.lsp ...
;; Loaded file power.lsp
T
[2]> (setf (long-float-digits) 256)
256
[3]> (setf a (+ 1 1L-25))
1.0000000000000000000000001L0
[4]> (power a (power 10 25))
2.71828182845904523536028733543857107480498532568559840479840654470561981531027L0
[5]>
```

Specifies that Clisp's LONG-FLOAT numbers are to have ⩾ 256 *binary* digits of precision.

1L-25 means the long-float with value 10^{-25}; this line sets **a** to the long-float with value $1 + 10^{-25}$.

This and earlier digits are the same as the corresponding digits of e.

Example of the Use of (floor n 2) as a Recursive Call Argument

Q. How can we write a recursive function **power** such that

(power z n) ⇒ z^n if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1 + 10^{-25})^{10^{25}}$?

- A solution is given by the function below:

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

- In public-key cryptography one often needs to perform ***modular exponentiation***, whose goal is to compute m^n mod k for integers m, n, and k (where $n \geq 0$ and $k > 0$). This can be done using a variant of the above function:

```
(defun power-mod (m n k) ; computes mⁿ mod k
  (cond ((zerop n) 1)
        (t (let ((X (power-mod m (floor n 2) k)))
             (cond ((evenp n) (mod (* X X) k))
                   (t (mod (* m X X) k)))))))
```

More Than One Formal Parameter of a Recursive Call May Have a Different Value from the Same Parameter of the Caller

- The **index** function in Assignment 5 illustrates this.
- Another illustration is provided by the exponentiation function below, which computes z^n using:

  $z^n = (z^2)^{n/2}$ if n is ***even***; $z^n = z*(z^2)^{\lfloor n/2 \rfloor}$ if n is ***odd***.

  Examples: $z^{12} = (z^2)^6$ and $z^{11} = z*(z^2)^5$.

```
(defun pwr (z n)
  (cond ((zerop n) 1)
        ((evenp n) (pwr (* z z) (/ n 2)))
        (t (* z (pwr (* z z) (floor n 2))))))
```

- The following function performs ***modular*** exponentiation in an analogous way:

```
(defun pwr-mod (m n k) ; computes m^n mod k
  (cond
    ((zerop n) 1)
    ((evenp n) (pwr-mod (mod (* m m) k) (/ n 2) k))
    (t (mod (* m (pwr-mod (mod (* m m) k) (floor n 2) k)) k))))
```

Using Different Recursive Calls for Different Argument Values

- MERGE-LISTS takes 2 arguments; ***each*** *argument value is assumed to be a proper* ***list of real numbers in ascending order***.
- (merge-lists L1 L2) returns a list that is equal to the list we would get if we sorted the list returned by (append L1 L2) into ascending order.

Two obvious recursive strategies to consider are:

1. Compute (merge-lists L1 L2) from (merge-lists (cdr L1) L2).
2. Compute (merge-lists L1 L2) from (merge-lists L1 (cdr L2)).

The function can indeed be written using these strategies, but *each of them is only good for* ***some*** *argument values*:

Example: L1 = (*2* **3 3 5 9 12**)   L2 = (*8* **10 11 14**)

(merge-lists (cdr L1) L2) should ⇒ (**3 3 5** *8* **9 10 11 12 14**)
(merge-lists L1 (cdr L2)) should ⇒ (*2* **3 3 5 9 10 11 12 14**)
(merge-lists L1 L2) should ⇒ (*2* **3 3 5** *8* **9 10 11 12 14**)

Getting (merge-lists L1 L2) from (merge-lists (cdr L1) L2), L1, L2 is ***easy***!

Getting (merge-lists L1 L2) from (merge-lists L1 (cdr L2)), L1, L2 is ***hard***!

Strategy **1** is right ***in this example***, because **(car L1) < (car L2)**.

Using Different Recursive Calls for Different Argument Values

- MERGE-LISTS takes 2 arguments; ***each*** *argument value is assumed to be a proper* ***list of real numbers in ascending order***.
- (merge-lists L1 L2) returns a list that is equal to the list we would get if we sorted the list returned by (append L1 L2) into ascending order.

Two obvious recursive strategies to consider are:

1. Compute (merge-lists L1 L2) from (merge-lists (cdr L1) L2).
2. Compute (merge-lists L1 L2) from (merge-lists L1 (cdr L2)).

Example: L1 = (2 3 3 5 9 12)   L2 = (8 10 11 14)

Strategy **1** is right ***in this example***, because **(car L1) < (car L2)**.

Example: L1 = (8 10 11 14)   L2 = (2 3 3 5 9 12)

Strategy **2** is right ***in this example***, because **(car L2) < (car L1)**.

Example: L1 = (2 8 10 11 14)   L2 = (2 3 3 5 9 12)

(merge-lists (cdr L1) L2) should ⇒ (2 3 3 5 8 9 10 11 12 14)
(merge-lists L1 (cdr L2)) should ⇒ (2 3 3 5 8 9 10 11 12 14)
(merge-lists L1 L2) should ⇒ (2 2 3 3 5 8 9 10 11 12 14)

(merge-lists (cdr L1) L2) and (merge-lists L1 (cdr L2)) are ***equal*** as **(car L2) = (car L1)**. Both strategies are good!

Using Different Recursive Calls for Different Argument Values

You can also write the UNREPEATED-ELTS and REPEATED-ELTS functions of Assignment 5 by using different recursive strategies for different argument values.

When f is either of these functions:

- Compute (f L) from (f (cdr L)) in certain non-base cases.
- Compute (f L) from (f (cddr L)) in other non-base cases.

Note: The MERGE-LISTS, UNREPEATED-ELTS, and REPEATED-ELTS functions are expected to make different direct recursive calls in different cases, but ***there should be no case in which in which these functions make more than one direct recursive call***!

Multiple Recursion

Example 10.1 We get a *flattened* form of a list if we ignore all but the initial opening and final closing parenthesis in the written representation of a list. The flattened form of

```
((a) ((b b)) (((c c c))))
```

From Sethi's book (and pp. 14 – 15 of the course reader).

is

```
(a b b c c c)
```

Function `flatten` constructs a flattened list by flattening the car and flattening the cdr of a list and appending the resulting sublists:

```
(define (flatten x)
  (cond ((null? x) x)
        ((not (pair? x)) (list x))
        (else (append (flatten (car x))
                      (flatten (cdr x)) ))))
```

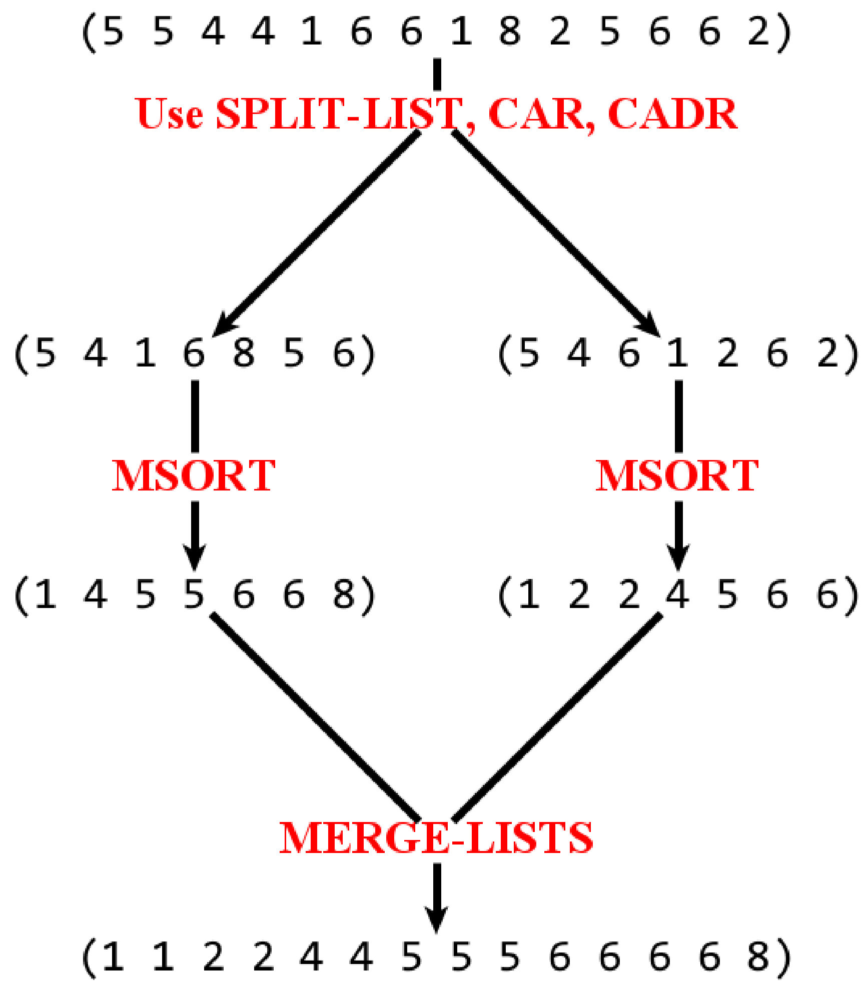
(pair? x) is Scheme's analog of (consp x): (pair? x) tests if x ⇒ a nonempty list.

Example: If x ⇒ (9 D (E (F) G)), then (flatten (car x)) ⇒ (9), (flatten (cdr x)) ⇒ (D E F G), and so (flatten x) ⇒ (9 D E F G).

Multiple Recursion (continued)

The sorting functions MSORT and QSORT of Assignment 5 should be doubly recursive.

- Here is a graphical illustration of how MSORT sorts the list (5 5 4 4 1 6 6 1 8 2 5 6 6 2) using ***two*** direct recursive calls of MSORT:

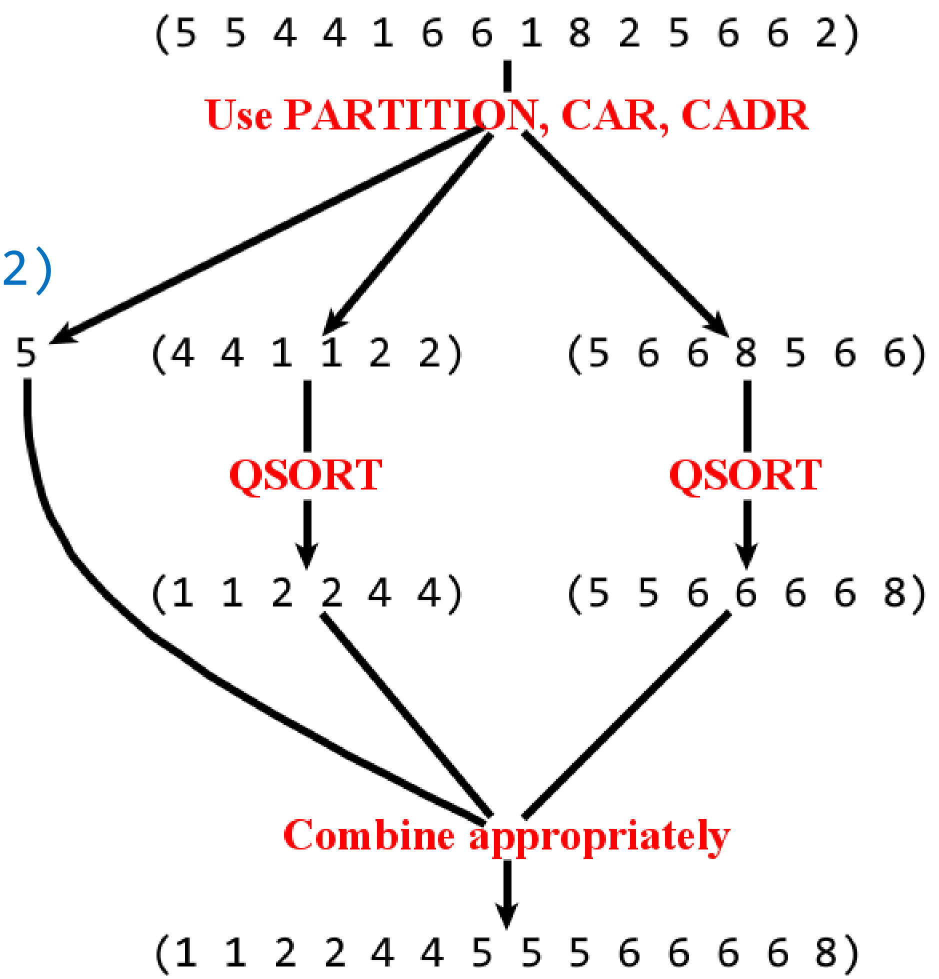


Multiple Recursion (continued)

The sorting functions MSORT and QSORT of Assignment 5 should be doubly recursive.

- Here is a graphical illustration of how QSORT sorts the list (5 5 4 4 1 6 6 1 8 2 5 6 6 2) using ***two*** direct recursive calls of QSORT:

- If p ⇒ a real no. and L ⇒ a list of real nos., then **(partition L p)** returns a list ((…) (…)) where (…) contains the elements of L that are < p, and (…) contains the elements of L that are ≥ p.



Examples of Functions That Take Functions as Arguments

Consider a function SIGMA such that, 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.]

For example, if

    g => the squaring function $x \mapsto x^2$

then we want

    (sigma g 2 5) => $2^2 + 3^2 + 4^2 + 5^2$ = 4+9+16+25 = 54.

Two questions are:

1. How do we *use* functions like SIGMA that take functions as arguments?

2. How do we *write* functions like SIGMA that take functions as arguments?

Question 1: How do we *use* functions like SIGMA that take functions as arguments?

To allow students to quickly test examples in clisp we first consider three *built-in* functions,

MAPCAR, **REMOVE-IF**, and **REMOVE-IF-NOT**,

that take functions as arguments.

However, functions like SIGMA that we may write ourselves can be called in a similar way!

NOTES: Scheme has built-in functions **map** and **filter** that are analogous to MAPCAR and REMOVE-IF-NOT (though **filter** isn't provided by kawa Scheme).

Problem 11 of Lisp Assignment 5 asks you to write a function **SUBSET** that behaves like the built-in function **REMOVE-IF-NOT**.

 From Touretzky's book.

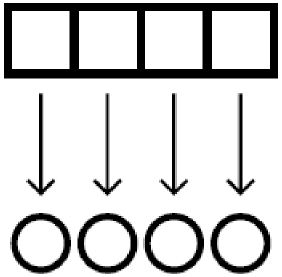
MAPCAR is the most frequently used applicative operator. It applies a function to each element of a list, one at a time, and returns a list of the results. Suppose we have written a function to square a single number. By itself, this function cannot square a list of numbers, because * doesn't work on lists.

```
(defun square (n) (* n n))

(square 3)  ⇒  9

(square '(1 2 3 4 5))  ⇒  Error! Wrong type input to *.
```

With MAPCAR we can apply SQUARE to each element of the list individually. To pass the SQUARE function as an input to MAPCAR, we quote it by writing #'SQUARE.

```
>  (mapcar #'square '(1 2 3 4 5))
(1 4 9 16 25)

> (mapcar #'square '(3 8 -3 5 2 10))
(9 64 9 25 4 100)
```

Here is a graphical description of the MAPCAR operator. As you can see, each element of the input list is mapped independently to a corresponding element in the output.



When MAPCAR is used on a list of length n, the resulting list also has exactly n elements. So if MAPCAR is used on the empty list, the result is the empty list.

```
(mapcar #'square '())  ⇒  nil
```

Some exercises from Touretzky's book:

7.1. Write an ADD1 function that adds one to its input. Then write an expression to add one to each element of the list (1 3 5 7 9).

7.2. Let the global variable DAILY-PLANET contain the following table:

```
((olsen jimmy 123-76-4535 cub-reporter)
 (kent  clark 089-52-6787 reporter)
 (lane  lois  951-26-1438 reporter)
 (white perry 355-16-7439 editor))
```

Each table entry consists of a last name, a first name, a social security number, and a job title. Use MAPCAR on this table to extract a list of social security numbers.

7.3. Write an expression to apply the ZEROP predicate to each element of the list (2 0 3 4 0 -5 -6). The answer you get should be a list of Ts and NILs.

7.4. Suppose we want to solve a problem similar to the preceding one, but instead of testing whether an element is zero, we want to test whether it is greater than five. We can't use > directly for this because > is a function of two inputs; MAPCAR will only give it one input. Show how first writing a one-input function called GREATER-THAN-FIVE-P would help.