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)))))))
```

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 $m^n \bmod k$ for integers *m*, *n*, and *k* (where $n \geq 0$ and $k > 0$). This can be done using a variant of the above function:

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 m^n 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

-
-
-

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.
-
-

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 $z^{11} = z*(z^2)^5$.

-

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))))))
```

-

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

Consider the MERGE-LISTS function of Assignment 5:

-
-

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

- MERGE-LISTS takes 2 arguments; ***each*** *argument value is assumed to be a proper* ***list of real numbers in ascending order***.
-

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

- 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.

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

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

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

- 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 .

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

- 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)).

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

- 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:

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:

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:

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 ⇒
(merge-lists L1 (cdr L2)) should ⇒
(merge-lists L1 L2) should ⇒

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 ⇒

(merge-lists L1 L2) should ⇒

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 ⇒

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**)

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 …

Getting (merge-lists L1 L2) from (merge-lists L1 (cdr L2)), L1, L2 is …

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 …

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***!

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 .

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)**.

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)**.

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**)

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)**.

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)

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 ⇒

(merge-lists L1 (cdr L2)) should ⇒

(merge-lists L1 L2) should ⇒

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 ⇒

(merge-lists L1 L2) should ⇒

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 ⇒

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)

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 (cdr 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

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:

-
-

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.
-

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:

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

A function is said to be ***multiply recursive*** if it sometimes makes more than one direct recursive call.

-

Multiple Recursion

A function is said to be ***multiply recursive*** if it sometimes makes more than one direct recursive call.

- Most multiply recursive functions never make more than two direct recursive calls. Such multiply recursive functions are often said to be ***doubly recursive***.

Multiple Recursion

A function is said to be ***multiply recursive*** if it sometimes makes more than one direct recursive call.

- Most multiply recursive functions never make more than two direct recursive calls. Such multiply recursive functions are often said to be ***doubly recursive***.

Sethi gives an example of such a function (written in Scheme) in Example 10.1 of the course reader:

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))))
```

is

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

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

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))))
```

is

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

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

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))))
```

is

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

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

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.

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 ⇒ (((A () B) (C)) D (E (F) G)), then (flatten (car x)) ⇒ , (flatten (cdr x)) ⇒ , and so (flatten x) ⇒ .

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 ⇒ (((A () B) (C)) D (E (F) G)), then (flatten (car x)) ⇒ (A B C), (flatten (cdr x)) ⇒ ,
and so (flatten x) ⇒ .

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 ⇒ (((A () B) (C)) D (E (F) G)), then (flatten (car x)) ⇒ (A B C), (flatten (cdr x)) ⇒ (D E F G), and so (flatten x) ⇒ .

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 ⇒ (((A () B) (C)) D (E (F) G)), then (flatten (car x)) ⇒ (A B C), (flatten (cdr x)) ⇒ (D E F G), and so (flatten x) ⇒ (A B C D E F G).

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 ⇒ , then
(flatten (car x)) ⇒ , (flatten (cdr x)) ⇒ ,
and so (flatten x) ⇒ .

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)) ⇒ , (flatten (cdr x)) ⇒ ,
and so (flatten x) ⇒ .

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)) ⇒ , and so (flatten x) ⇒ .

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) ⇒ .

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.

Multiple Recursion (continued)

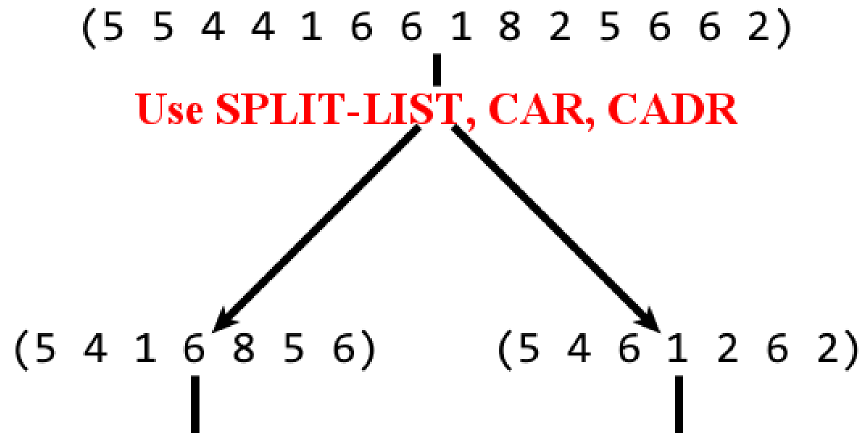
The sorting functions MSORT and QSORT of Assignment 5 should be doubly recursive.

- Below 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.

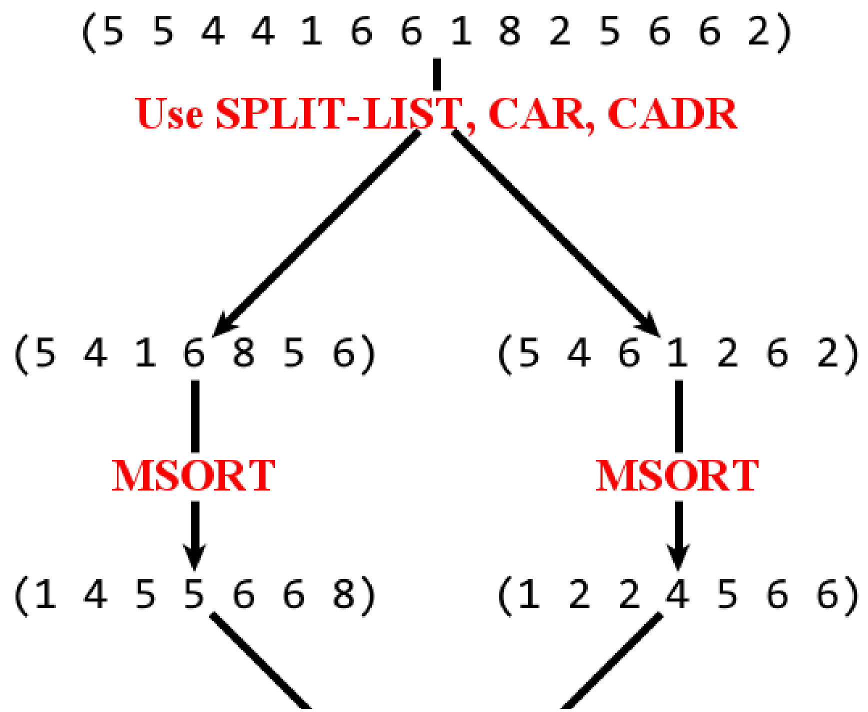
- Here is a graphical illustration of how MSORT sorts the list (5 5 4 4 1 6 6 1 8 2 5 6 6 2) using ***two*** direct recursive calls of MSORT:



Multiple Recursion (continued)

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

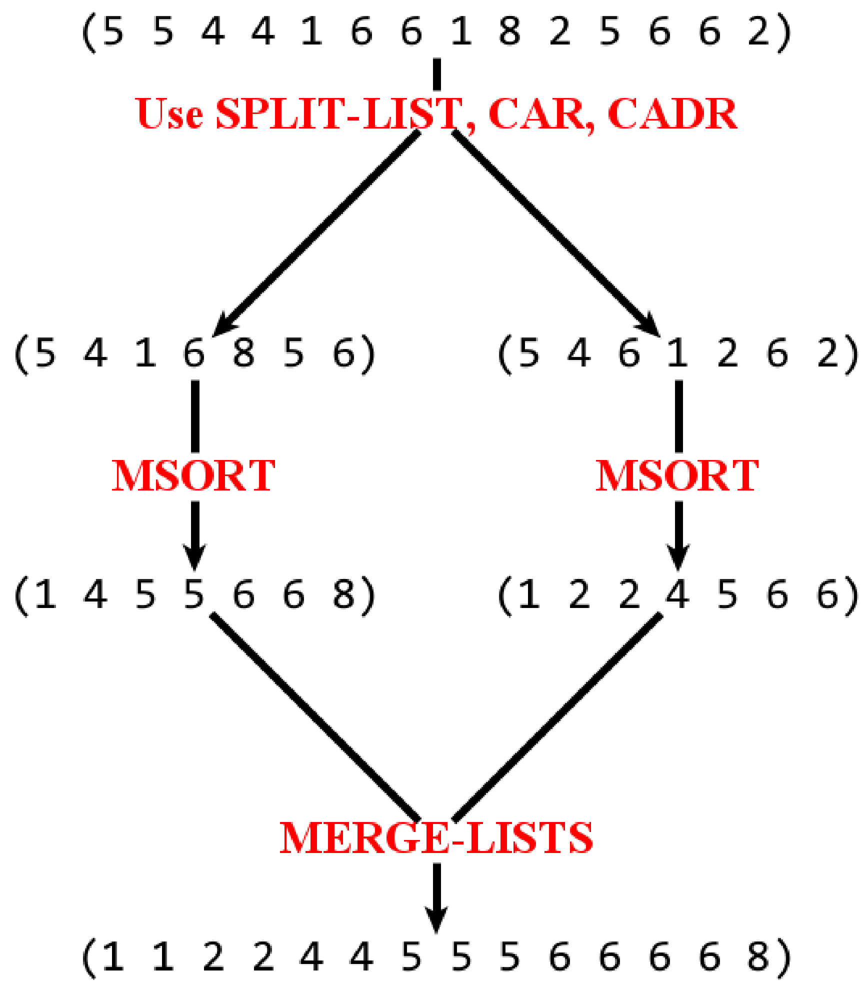
- Here is a graphical illustration of how MSORT sorts the list (5 5 4 4 1 6 6 1 8 2 5 6 6 2) using ***two*** direct recursive calls of MSORT:



Multiple Recursion (continued)

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

- Here is a graphical illustration of how 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.

Multiple Recursion (continued)

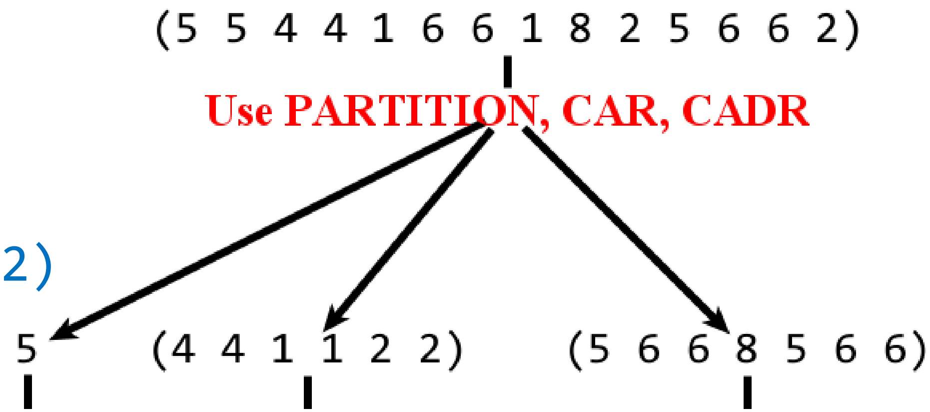
The sorting functions MSORT and QSORT of Assignment 5 should be doubly recursive.

- Below 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:

Multiple Recursion (continued)

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

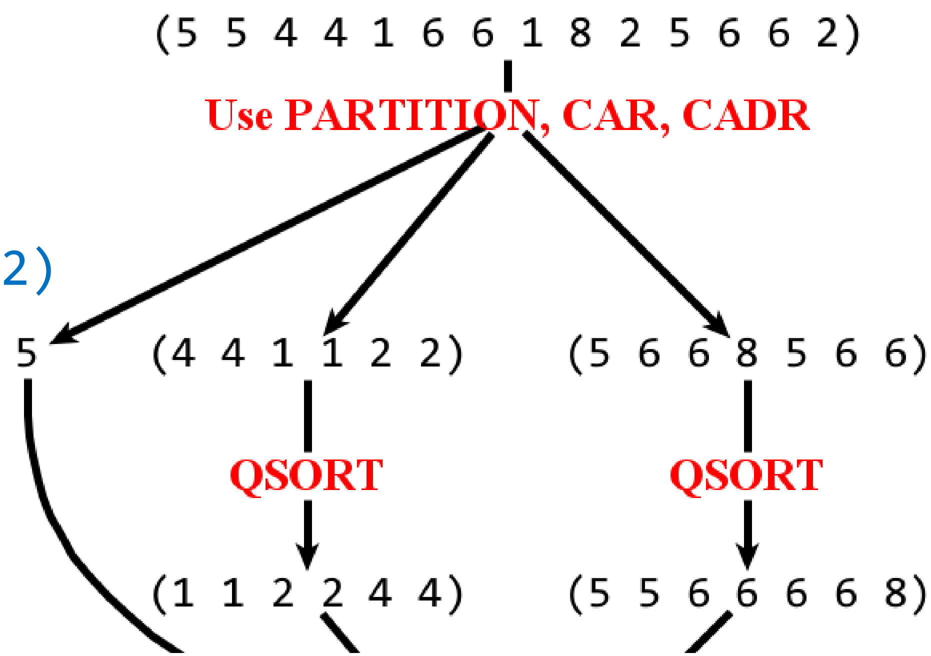
- Here is a graphical illustration of how QSORT sorts the list (5 5 4 4 1 6 6 1 8 2 5 6 6 2) using ***two*** direct recursive calls of QSORT:



- 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.

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:

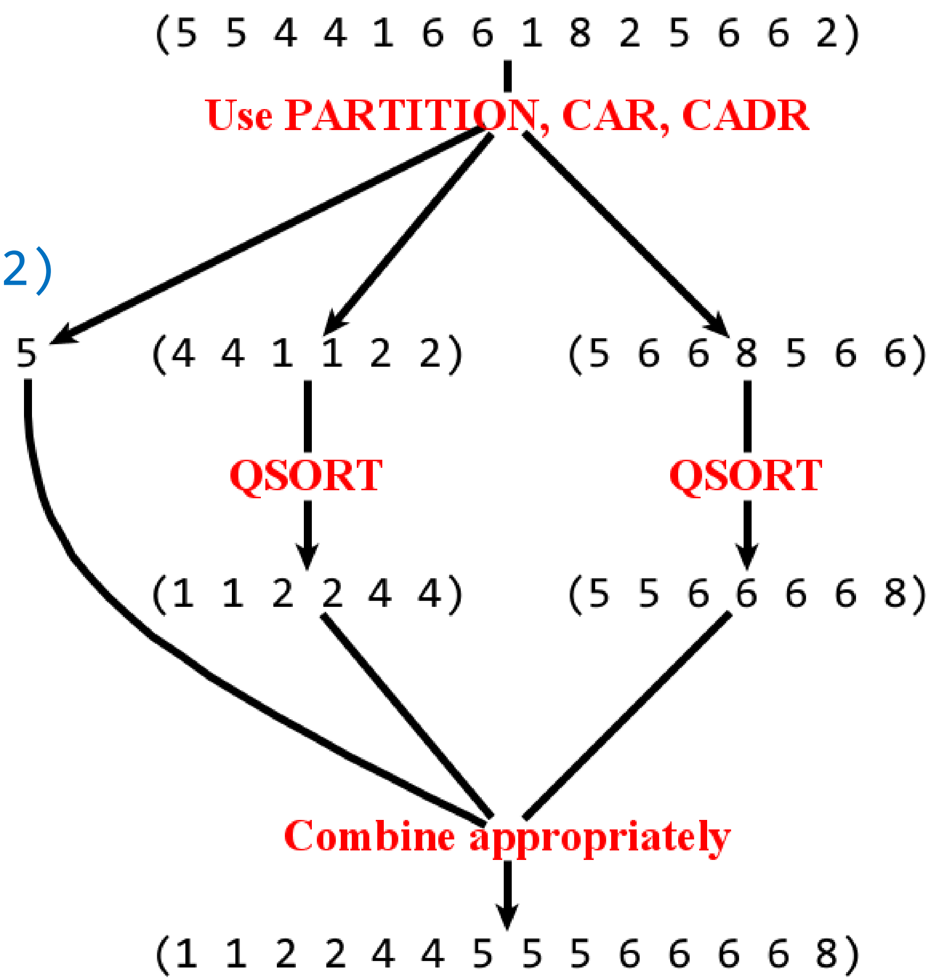
- If p ⇒ a real no. and L ⇒ a list of real nos., then **(partition L p)** returns a list **(**(…) (…)**)** where **(…)** contains the elements of L that are **<** p, and **(…)** contains the elements of L that are **≥** p.

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.



Functions That Take Functions as Arguments

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).
```

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.]
```

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 ↦ x²
then we want
     (sigma g 2 5) =>
```

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:

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?

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 ↦ x²
then we want
     (sigma g 2 5) => 2² + 3² + 4² + 5² = 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?

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!

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:

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**.

7.3 THE MAPCAR OPERATOR

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.

7.3 THE MAPCAR OPERATOR

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))
```

 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)
```

 From Touretzky's book.

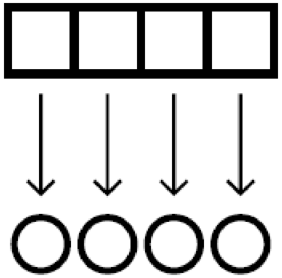
MAPCAR is the most frequently used applicative operator. It applies a function to each element of a list, one at a time, and returns a list of the results. Suppose we have written a function to square a single number. By itself, this function cannot square a list of numbers, because * doesn't work on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

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.

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.

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.

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.*

7.5 LAMBDA EXPRESSIONS

From Touretzky's book.

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.*

7.5 LAMBDA EXPRESSIONS

From Touretzky’s book.

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))
```

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*.

7.5 LAMBDA EXPRESSIONS

From Touretzky's book.

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!

From sec. 7.5 of Touretzky's book:

Lambda expressions are especially useful for synthesizing one-input functions from related functions of two inputs. For example, suppose we wanted to multiply every element of a list by 10. We might be tempted to write something like:

From sec. 7.5 of Touretzky's book:

Lambda expressions are especially useful for synthesizing one-input functions from related functions of two inputs. For example, suppose we wanted to multiply every element of a list by 10. We might be tempted to write something like:

```
(mapcar #'* '(1 2 3 4 5))
```

but where is the 10 supposed to go? The * function needs two inputs, but MAPCAR is only going to give it one.

From sec. 7.5 of Touretzky's book:

Lambda expressions are especially useful for synthesizing one-input functions from related functions of two inputs. For example, suppose we wanted to multiply every element of a list by 10. We might be tempted to write something like:

```
(mapcar #'* '(1 2 3 4 5))
```

but where is the 10 supposed to go? The * function needs two inputs, but MAPCAR is only going to give it one. The correct way to solve this problem is to write a lambda expression of *one* input that multiplies its input by 10. Then we can feed the lambda expression to MAPCAR.

```
> (mapcar #'(lambda (n) (* n 10)) '(1 2 3 4 5))
(10 20 30 40 50)
```

From sec. 7.5 of Touretzky's book:

Lambda expressions are especially useful for synthesizing one-input functions from related functions of two inputs. For example, suppose we wanted to multiply every element of a list by 10. We might be tempted to write something like:

```
(mapcar #'* '(1 2 3 4 5))
```

but where is the 10 supposed to go? The * function needs two inputs, but MAPCAR is only going to give it one. The correct way to solve this problem is to write a lambda expression of *one* input that multiplies its input by 10. Then we can feed the lambda expression to MAPCAR.

```
> (mapcar #'(lambda (n) (* n 10)) '(1 2 3 4 5))
(10 20 30 40 50)
```

Here is another example of the use of MAPCAR along with a lambda expression. We will turn each element of a list of names into a list (HI THERE *name*).

From sec. 7.5 of Touretzky's book:

Lambda expressions are especially useful for synthesizing one-input functions from related functions of two inputs. For example, suppose we wanted to multiply every element of a list by 10. We might be tempted to write something like:

```
(mapcar #'* '(1 2 3 4 5))
```

but where is the 10 supposed to go? The * function needs two inputs, but MAPCAR is only going to give it one. The correct way to solve this problem is to write a lambda expression of *one* input that multiplies its input by 10. Then we can feed the lambda expression to MAPCAR.

```
> (mapcar #'(lambda (n) (* n 10)) '(1 2 3 4 5))
(10 20 30 40 50)
```

Here is another example of the use of MAPCAR along with a lambda expression. We will turn each element of a list of names into a list (HI THERE *name*).

```
> (mapcar #'(lambda (x) (list 'hi 'there x))
          '(joe fred wanda))
((HI THERE JOE) (HI THERE FRED) (HI THERE WANDA))
```

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))
```

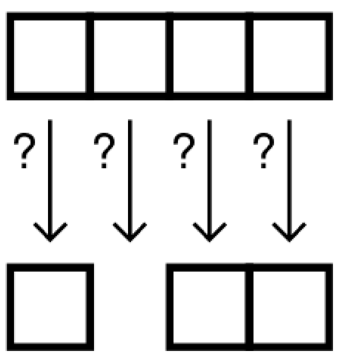
7.8 REMOVE-IF AND REMOVE-IF-NOT

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.

From Touretzky's Book

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

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.

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):

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.

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