From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values.

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

```
(defun price-change (old new)
  (let* ((diff (- new old))
```

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

```
(defun price-change (old new)
  (let* ((diff (- new old))
         (proportion (/ diff old))
```

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

```
(defun price-change (old new)
  (let* ((diff (- new old))
         (proportion (/ diff old))
         (percentage (* proportion 100.0)))
```

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

```
(defun price-change (old new)
  (let* ((diff (- new old))
         (proportion (/ diff old))
         (percentage (* proportion 100.0)))
    (list 'widgets 'changed 'by percentage
          'percent)))
```

From p. 144 of Touretzky:

For example, suppose we want a function that computes the percent change in the price of widgets given the old and new prices as input. Our function must compute the difference between the two prices, then divide this difference by the old price to get the proportional change in price, and then multiply that by 100 to get the percent change. We can use local variables named DIFF, PROPORTION, and PERCENTAGE to hold these values. We use LET* instead of LET because these variables must be created one at a time, since each depends on its predecessor.

```
(defun price-change (old new)
  (let* ((diff (- new old))
         (proportion (/ diff old))
         (percentage (* proportion 100.0)))
    (list 'widgets 'changed 'by percentage
          'percent)))

> (price-change 1.25 1.35)
(WIDGETS CHANGED BY 8.0 PERCENT)
```

From p. 145 of Touretzky:

A common programming error is to use LET when LET* is required. Consider the following FAULTY-SIZE-RANGE function. It uses MAX and MIN to find the largest and smallest of a group of numbers. MAX and MIN are built in to Common Lisp; they both accept one or more inputs. The extra 1.0 argument to / is used to force the result to be a floating point number rather than a ratio.

```
(defun faulty-size-range (x y z)
  (let ((biggest (max x y z))
        (smallest (min x y z))
        (r (/ biggest smallest 1.0)))
    (list 'factor 'of r)))
```

From p. 145 of Touretzky:

A common programming error is to use LET when LET* is required. Consider the following FAULTY-SIZE-RANGE function. It uses MAX and MIN to find the largest and smallest of a group of numbers. MAX and MIN are built in to Common Lisp; they both accept one or more inputs. The extra 1.0 argument to / is used to force the result to be a floating point number rather than a ratio.

```
(defun faulty-size-range (x y z)
  (let ((biggest (max x y z))
        (smallest (min x y z))
        (r (/ biggest smallest 1.0)))
    (list 'factor 'of r)))
```

```
> (faulty-size-range 35 87 4)
Error in function SIZE-RANGE:
  BIGGEST unassigned variable.
```

From p. 145 of Touretzky:

A common programming error is to use LET when LET* is required. Consider the following FAULTY-SIZE-RANGE function. It uses MAX and MIN to find the largest and smallest of a group of numbers. MAX and MIN are built in to Common Lisp; they both accept one or more inputs. The extra 1.0 argument to / is used to force the result to be a floating point number rather than a ratio.

```
(defun faulty-size-range (x y z)
  (let ((biggest (max x y z))
        (smallest (min x y z))
        (r (/ biggest smallest 1.0)))
    (list 'factor 'of r)))

> (faulty-size-range 35 87 4)
Error in function SIZE-RANGE:
  BIGGEST unassigned variable.
```

The problem is that the expression (/ BIGGEST SMALLEST 1.0) is being evaluated in a lexical context that does not include these variables. Therefore the symbol BIGGEST is interpreted as a reference to a global variable

The following explanation of the difference between LET and LET* appears on p. 391 of Sethi (p. 7 of the course reader):

A sequential variant of the `let` construct is written with keyword `let*`. Unlike `let`, which evaluates all the expressions $E_1, E_2, \ldots, E_k$ before binding any of the variables, `let*` binds x_i to the value of E_i before E_{i+1} is evaluated. The syntax is

(let* ((x_1 E_1) (x_2 E_2) $\cdots$ (x_k E_k)) F)

The distinction between `let` and `let*` can be seen from the responses in

The following explanation of the difference between LET and LET* appears on p. 391 of Sethi (p. 7 of the course reader):

A sequential variant of the `let` construct is written with keyword `let*`. Unlike `let`, which evaluates all the expressions $E_1, E_2, \ldots, E_k$ before binding any of the variables, `let*` binds x_i to the value of E_i before E_{i+1} is evaluated. The syntax is

$$\texttt{(let* ((}x_1\ E_1\texttt{)}\ \texttt{(}x_2\ E_2\texttt{)}\ \cdots\ \texttt{(}x_k\ E_k\texttt{))}\ F\texttt{)}$$

The distinction between `let` and `let*` can be seen from the responses in

Use (**setf** x 0) here in Common Lisp.

~~(define x 0)~~

~~x~~

```
(let ((x 2) (y x)) y)     ; bind y before redefining x

(let* ((x 2) (y x)) y)    ; bind y after redefining x
```

The following explanation of the difference between LET and LET* appears on p. 391 of Sethi (p. 7 of the course reader):

A sequential variant of the `let` construct is written with keyword `let*`. Unlike `let`, which evaluates all the expressions $E_1, E_2, \ldots, E_k$ before binding any of the variables, `let*` binds x_i to the value of E_i before E_{i+1} is evaluated. The syntax is

`(let* ((`x_1 E_1`) (`x_2 E_2`)` $\cdots$ `(`x_k E_k`))` F`)`

The distinction between `let` and `let*` can be seen from the responses in

Use (**setf** x 0) here in Common Lisp.

~~(define x 0)~~

~~x~~

```
(let ((x 2) (y x)) y)     ; bind y before redefining x
  0
(let* ((x 2) (y x)) y)    ; bind y after redefining x
```

The following explanation of the difference between LET and LET* appears on p. 391 of Sethi (p. 7 of the course reader):

A sequential variant of the `let` construct is written with keyword `let*`. Unlike `let`, which evaluates all the expressions $E_1, E_2, \ldots, E_k$ before binding any of the variables, `let*` binds x_i to the value of E_i before E_{i+1} is evaluated. The syntax is

$$(\texttt{let*}\ ((x_1\ E_1)\ (x_2\ E_2)\ \cdots\ (x_k\ E_k))\ F)$$

The distinction between `let` and `let*` can be seen from the responses in

Use **(setf x 0)** here in Common Lisp.

```
~~(define x 0)~~
   ~~x~~
(let ((x 2) (y x)) y)      ; bind y before redefining x
   0
(let* ((x 2) (y x)) y)     ; bind y after redefining x
   2
```

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h* 3 4 7):

(h* 3 4 7) ⇒

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*,
we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h* 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**

the LET*'s local x ⇒

the LET*'s local y ⇒

the LET*'s local z ⇒

LET* ⇒

x ⇒

(h* 3 4 7) ⇒

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Evaluation of (h* 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET*'s local x ⇒ **2**
the LET*'s local y ⇒
the LET*'s local z ⇒
LET* ⇒
x ⇒

(h* 3 4 7) ⇒

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Evaluation of (h* 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**
the LET*'s local x ⇒ **2**
the LET*'s local y ⇒ **4**
the LET*'s local z ⇒
LET* ⇒
x ⇒

(h* 3 4 7) ⇒

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h* 3 4 7):

w ⇒ **3**   x ⇒ **4**   y ⇒ **7**
the LET*'s local x ⇒ **2**
the LET*'s local y ⇒ **4**
the LET*'s local z ⇒ **6**
LET* ⇒
x ⇒

(h* 3 4 7) ⇒

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h* 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET*'s local x ⇒ **2**
the LET*'s local y ⇒ **4**
the LET*'s local z ⇒ **6**
LET* ⇒ **(2+4+6)/3 = 4**
x ⇒

(h* 3 4 7) ⇒

Recall:

Evaluation of (h 3 4 7):

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET's local x ⇒ **2**

the LET's local y ⇒ **8**

the LET's local z ⇒ **11**

LET ⇒ **(2+8+11)/3 = 7**

x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*, we get:

Evaluation of (h* 3 4 7):

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

w ⇒ **3** x ⇒ **4** y ⇒ **7**

the LET*'s local x ⇒ **2**

the LET*'s local y ⇒ **4**

the LET*'s local z ⇒ **6**

LET* ⇒ **(2+4+6)/3 = 4**

x ⇒ **4**

(h* 3 4 7) ⇒

Recall:

```
(defun h (w x y)
  (+ (let ((x (sqrt x))
           (y (* x 2))
           (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET's local x ⇒ **2**
the LET's local y ⇒ **8**
the LET's local z ⇒ **11**
LET ⇒ **(2+8+11)/3 = 7**
x ⇒ **4**

(h 3 4 7) ⇒ **7+4 = 11**

Replacing let with let*, we get:

```
(defun h* (w x y)
  (+ (let* ((x (sqrt x))
            (y (* x 2))
            (z (+ x y)))
       (/ (+ x y z) w))
     x))
```

Evaluation of (h* 3 4 7):

w ⇒ **3** x ⇒ **4** y ⇒ **7**
the LET*'s local x ⇒ **2**
the LET*'s local y ⇒ **4**
the LET*'s local z ⇒ **6**
LET* ⇒ **(2+4+6)/3 = 4**
x ⇒ **4**

(h* 3 4 7) ⇒ **4+4 = 8**

Recursive Functions

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

-

-

-

Example

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
-
-

Example

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.
-

Example

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.
- In more sophisticated recursion we may occasionally relax this "*smaller in size than x*" condition, but then we have to carefully check that our function terminates!

Example

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.
- In more sophisticated recursion we may occasionally relax this "*smaller in size than x*" condition, but then we have to carefully check that our function terminates!

Example Write a function **length-of** such that:

If l ⇒ a proper list, then
(length-of l) ⇒ the length of l.

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then
(length-of l) ⇒ the length of l.

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
         ))
```

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x*.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

-

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x*.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

- If L ⇒ NIL, this violates the "call the supposedly already written f *with an argument value that is … **smaller***" condition.

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

To make our function good **even when** L **⇒ NIL**, we add a case:

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If L ⇒ a proper list, then (length-of L) ⇒ the length of L.

Assuming **length-of** has already been written, here is a function that works ***provided*** L ⇒ a **nonempty** list:

```
(defun my-length-of (L)
  (let ((X (length-of (cdr L))))
    (+ X 1)))
```

To make our function good **even when** L **⇒ NIL**, we add a case:

```
(defun better-my-length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works:

```
(defun better-my-length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works:

```
(defun better-my-length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works:

```
(defun better-my-length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

But this still assumes **length-of** has ***already*** been written.

Q. How can we write **length-of**?

A.

Writing Recursive Functions That Take 1 Argument, Which is a Proper List or a Nonnegative Integer

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

Assuming **length-of** has already been written, here is a function that works:

```
(defun better-my-length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

But this still assumes **length-of** has ***already*** been written.

Q. How can we write **length-of**?

A. We simply ***rename*** **better-my-length-of** to **length-of!**

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

Q. How can we write **length-of**?

A. We simply ***rename*** **better-my-length-of** to **length-of!**

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L)
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

Q. How can we write **length-of**?

A. We simply ***rename*** **better-my-length-of** to **length-of!**

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

Q. How can we write **length-of**?

A. We simply ***rename*** **better-my-length-of** to **length-of!**

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received.*

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received.*

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received.*
-
-
-

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is **smaller** than the argument value it received.*
- ***If*** a recursive call **(length-of (cdr L))** returns the right result, ***then*** the call **(length-of L)** returns the right result.
-
-

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun better-my-length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received.*
- ***If*** a recursive call **(length-of (cdr L))** returns the right result, ***then*** the call **(length-of L)** returns the right result.
- So, for all *n* > 0, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
-

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun better-my-length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- This definition of **length-of** is ***not*** circular, because when **length-of** calls itself it always *passes an argument value that is **smaller** than the argument value it received.*
- ***If*** a recursive call **(length-of (cdr L))** returns the right result, ***then*** the call **(length-of L)** returns the right result.
- So, for all $n > 0$, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length $< n$, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- **Example:** ***If*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 0, 1, 2, or 3, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 4.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all $n > 0$, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- **Example:** ***If*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 0, 1, 2, or 3, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 4.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all $n > 0$, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length $< n$, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length n.
- **Example:** ***If*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 0, 1, 2, or 3, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 4.

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all *n* > 0, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- **Example:** ***If*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 0, 1, 2, or 3, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 4.
- (length-of *l*) returns the right result (i.e., 0) when *l* ⇒ a list of length 0.

∴

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all *n* > 0, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- **Example:** ***If*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 0, 1, 2, or 3, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length 4.
- (length-of *l*) returns the right result (i.e., 0) when *l* ⇒ a list of length 0.

∴ If *l* ⇒ a proper list of any length,
then (length-of *l*) ⇒ the right result (i.e., *l*'s length).

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all *n* > 0, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.

- (length-of *l*) returns the right result (i.e., 0) when *l* ⇒ a list of length 0.

∴ If *l* ⇒ a proper list of any length,
then (length-of *l*) ⇒ the right result (i.e., *l*'s length).

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- For all $n > 0$, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- (length-of *l*) returns the right result (i.e., 0) when *l* ⇒ a list of length 0.

∴ If *l* ⇒ a proper list of any length,
then (length-of *l*) ⇒ the right result (i.e., *l*'s length).

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

(defun ~~better-my-~~length-of (L)
  (if (null L) *; base case, where there's no recursive call*
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))

- For all *n* > 0, ***if*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length < *n*, ***then*** (length-of *l*) returns the right result when *l* ⇒ a proper list of length *n*.
- (length-of *l*) returns the right result (i.e., 0) when *l* ⇒ a list of length 0.

∴ If *l* ⇒ a proper list of any length, then (length-of *l*) ⇒ the right result (i.e., *l*'s length).

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the* LET*,*** ***because its local variable*** X ***is never used more than once.***

We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the LET,*** ***because its local variable*** X ***is never used more than once.***

  We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- Although this function is correct as written, *we can improve / simplify the definition by **eliminating the LET**, **because its local variable** X **is never used more than once**.* We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun better-my-length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the LET*****,** ***because its local variable*** X ***is never used more than once*****.** We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

```
(defun length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X (length-of (cdr L)) 1))))
```

Example Write a function **length-of** such that:

If l ⇒ a proper list, then (length-of l) ⇒ the length of l.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the*** **LET**, ***because its local variable*** X ***is never used more than once***. We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

```
(defun length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (+ (length-of (cdr L)) 1)))
```

-

Example Write a function **length-of** such that:

If L ⇒ a proper list, then (length-of L) ⇒ the length of L.

```
(defun ~~better-my-~~length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (let ((X (length-of (cdr L))))
        (+ X 1))))
```

- Although this function is correct as written, *we can improve / simplify the definition by eliminating the LET, because its local variable X is never used more than once.* We then replace the X in **(+ X 1)** with **(length-of (cdr L))**:

```
(defun length-of (L)
  (if (null L) ; base case, where there's no recursive call
      0
      (+ (length-of (cdr L)) 1)))
```

- We've given a written explanation of a possible thought process that leads to this definition, but an experienced Lisp programmer would likely code simple definitions like this one without giving any explanation!

Example Write a function **factorial** such that:

If *n* ⇒ *a non-negative integer*, *then* (factorial *n*) ⇒ *n*!.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

Recall the following rules for writing recursive functions of 1 argument, which is a proper List or a nonnegative integer:

-

-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

Recall the following rules for writing recursive functions of 1 argument, which is a proper List or a nonnegative integer:

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

Recall the following rules for writing recursive functions of 1 argument, which is a proper List or a nonnegative integer:

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

Recall the following rules for writing recursive functions of 1 argument, which is a proper List or a nonnegative integer:

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x*.

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
          ))
```

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

Recall the following rules for writing recursive functions of 1 argument, which is a proper List or a nonnegative integer:

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x*.

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
           ))
```

- **We use the fact that:** **n * (n-1)! = n!**
  For example: **5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!**

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

Recall the following rules for writing recursive functions of 1 argument, which is a proper List or a nonnegative integer:

- When writing a recursive function f, we can first suppose a function f that correctly solves the same problem has ***already*** been written.
- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Assuming ***factorial*** *has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- **We use the fact that:** **n * (n-1)! = n!**
  For example: **5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!**

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x*.

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- **We use the fact that:** **n * (n-1)! = n!**
  For example: **5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!**

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size than x*.

Assuming ***factorial*** *has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- **We use the fact that:** **n * (n-1)! = n!**

  For example: **5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!**

-
-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x.*

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- **We use the fact that:** **n * (n-1)! = n!**
  For example: **5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!**
- Importantly, **n * (n-1)! = n!** holds even when n = 1, as 0! = 1.
-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

- Our own version of f can call the supposedly already written f; but when our version is called with an argument value *x*, it is only allowed to call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size than x.*

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- **We use the fact that:** **n * (n-1)! = n!**
  For example: **5 * 4! = 5 * 4 * 3 * 2 * 1 = 5!**
- Importantly, **n * (n-1)! = n!** holds even when n = 1, as 0! = 1.
- If n ⇒ 0, the above definition ***violates*** the "call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size*" condition, because **(- n 1)** is ***not*** a valid argument value for **factorial** if n ⇒ 0.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

Assuming ***factorial*** *has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- If n ⇒ 0, the above definition ***violates*** the "call the supposedly already written f *with an argument value that is* ***valid*** *for* f *and* ***smaller*** *in size*" condition, because **(- n 1)** is ***not*** a valid argument value for **factorial** if n ⇒ 0.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- If n ⇒ 0, the above definition ***violates*** the "call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size*" condition, because **(- n 1)** is ***not*** a valid argument value for **factorial** if n ⇒ 0.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

*Assuming **factorial** has already been written correctly*, here is a function that works ***provided*** n ⇒ a **nonzero** integer:

```
(defun my-factorial (n)
  (let ((X (factorial (- n 1))))
    (* n X)))
```

- If n ⇒ 0, the above definition ***violates*** the "call the supposedly already written f *with an argument value that is **valid** for* f *and **smaller** in size*" condition, because **(- n 1)** is ***not*** a valid argument value for **factorial** if n ⇒ 0.
- To make our function good **even when** n **⇒** 0, we add a case:

```
(defun better-my-factorial (n)
  (if (zerop n)
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

*Assuming **factorial** has already been written correctly*, here is a function that works:

```
(defun better-my-factorial (n)
  (if (zerop n)
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

*Assuming **factorial** has already been written correctly*, here is a function that works:

```
(defun better-my-factorial (n)
  (if (zerop n)
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

Assuming ***factorial*** *has already been written correctly*, here is a function that works:

```
(defun better-my-factorial (n)
  (if (zerop n)
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

But this still assumes **factorial** has ***already*** been written.

Q. How can we write **factorial**?

A. We simply ***rename*** **better-my-factorial** to **factorial!**

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n)
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

Q. How can we write **factorial**?

A. We simply ***rename*** **better-my-factorial** to **factorial!**

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

(defun ~~better-my-~~factorial (n)
** (if (zerop n)**
**   1**
**   (let ((X (factorial (- n 1))))**
**   (* n X))))**

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- This definition of **factorial** is ***not*** circular, because when **factorial** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received.*
-
-
-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- This definition of **factorial** is ***not*** circular, because when **factorial** calls itself it always *passes an argument value that is **smaller** than the argument value it received.*
- ***If*** a recursive call **(factorial (- n 1))** returns the right result, ***then*** the call **(factorial n)** returns the right result.
-
-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- This definition of **factorial** is ***not*** circular, because when **factorial** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received*.
- ***If*** a recursive call **(factorial (- n 1))** returns the right result, ***then*** the call **(factorial n)** returns the right result.
- So, for all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

`(defun` ~~`better-my-`~~`factorial (n)`
`  (if (zerop n) ; base case, where there's no recursive call`
`      1`
`      (let ((X (factorial (- n 1))))`
`        (* n X))))`

- This definition of **factorial** is ***not*** circular, because when **factorial** calls itself it always *passes an argument value that is* ***smaller*** *than the argument value it received*.
- ***If*** a recursive call **(factorial (- n 1))** returns the right result, ***then*** the call **(factorial n)** returns the right result.
- So, for all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- **Example:** ***If*** (factorial *i*) returns the right result when *i* ⇒ 0, 1, 2, or 3, ***then*** (factorial *i*) also returns the right result when *i* ⇒ 4.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers k, ***if*** (factorial i) returns the right result whenever i ⇒ a nonnegative integer < k, ***then*** (factorial i) also returns the right result when i ⇒ k.
- **Example:** ***If*** (factorial i) returns the right result when i ⇒ 0, 1, 2, or 3, ***then*** (factorial i) also returns the right result when i ⇒ 4.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- **Example:** ***If*** (factorial *i*) returns the right result when *i* ⇒ 0, 1, 2, or 3, ***then*** (factorial *i*) also returns the right result when *i* ⇒ 4.

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- **Example:** ***If*** (factorial *i*) returns the right result when *i* ⇒ 0, 1, 2, or 3, ***then*** (factorial *i*) also returns the right result when *i* ⇒ 4.
- (factorial *i*) returns the right result (i.e., 1) when *i* ⇒ 0.

∴ If *i* ⇒ any nonnegative integer,
then (factorial *i*) ⇒ the right result (i.e., *i*!).

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.

- (factorial *i*) returns the right result (i.e., 1) when *i* ⇒ 0.

∴ If *i* ⇒ any nonnegative integer,
then (factorial *i*) ⇒ the right result (i.e., *i*!).

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

```
(defun better-my-factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- (factorial *i*) returns the right result (i.e., 1) when *i* ⇒ 0.

∴ If *i* ⇒ any nonnegative integer,
then (factorial *i*) ⇒ the right result (i.e., *i*!).

-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- For all positive integers *k*, ***if*** (factorial *i*) returns the right result whenever *i* ⇒ a nonnegative integer < *k*, ***then*** (factorial *i*) also returns the right result when *i* ⇒ *k*.
- (factorial *i*) returns the right result (i.e., 1) when *i* ⇒ 0.

∴ If *i* ⇒ any nonnegative integer,
then (factorial *i*) ⇒ the right result (i.e., *i*!).

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the*** **LET**, ***because its local variable*** X ***is never used more than once***.

We then replace the X in **(* n X)** with **(factorial (- n 1))**:

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the LET, because its local variable*** X ***is never used more than once.***

  We then replace the X in **(* n X)** with **(factorial (- n 1))**:

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the LET, because its local variable*** X ***is never used more than once.*** We then replace the X in **(* n X)** with **(factorial (- n 1))**:

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- Although this function is correct as written, *we can improve / simplify the definition by* ***eliminating the* LET*,*** ***because its local variable*** X ***is never used more than once***. We then replace the X in **(* n X)** with **(factorial (- n 1))**:

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      ~~(let ((X (factorial (- n 1))))~~
        (* n ~~X~~ (factorial (- n 1)))~~)~~))
```

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial n) ⇒ n!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- Although this function is correct as written, *we can improve / simplify the definition by __eliminating the LET__, **because its local variable** X **__is never used more than once__.*** We then replace the X in **(* n X)** with **(factorial (- n 1))**:

```
(defun factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (* n (factorial (- n 1)))))
```

-

Example Write a function **factorial** such that:

If n ⇒ a non-negative integer, then (factorial *n*) ⇒ *n*!.

```
(defun ~~better-my-~~factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (let ((X (factorial (- n 1))))
        (* n X))))
```

- Although this function is correct as written, *we can improve / simplify the definition by **eliminating the LET**, **because its local variable** X **is never used more than once**.* We then replace the X in **(* n X)** with **(factorial (- n 1))**:

```
(defun factorial (n)
  (if (zerop n) ; base case, where there's no recursive call
      1
      (* n (factorial (- n 1)))))
```

- As in the case of **length-of**, we've given a written explanation of a possible thought process that leads to this definition, but a Lisp programmer would likely code simple definitions like these without giving any explanation!

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written in the above way.
- The resulting definition will then have the following form (before possible elimination of the LET):

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written in the above way.
- The resulting definition will then have the following form (before possible elimination of the LET):

```
(defun f (e)
  (if (null e)

      (let ((X (f (cdr e))))

                                                          )))
```

OR

```
(defun f (e)
  (if (zerop e)

      (let ((X (f (- e 1))))

                                                          )))
```

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written in the above way.
- The resulting definition will then have the following form (before possible elimination of the LET):

```
(defun f (e)
  (if (null e)
      value of (f nil)
      (let ((X (f (cdr e))))
                                  )))
```

OR

```
(defun f (e)
  (if (zerop e)
      value of (f 0)
      (let ((X (f (- e 1))))
                                  )))
```

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written in the above way.
- The resulting definition will then have the following form (before possible elimination of the LET):

```
(defun f (e)
  (if (null e)
      value of (f nil)
      (let ((X (f (cdr e))))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e )))
```

OR

```
(defun f (e)
  (if (zerop e)
      value of (f 0)
      (let ((X (f (- e 1))))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e )))
```

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written as follows:

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written as follows:

```
(defun f (e)
  (if (null e) or (zerop e)
      value of (f nil) or (f 0)
      (let ((X (f (cdr e)) or (f (- e 1)) ))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e )))
```

-
-
-

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written as follows:

```
(defun f (e)
  (if (null e) or (zerop e)
      value of (f nil) or (f 0)
      (let ((X (f (cdr e)) or (f (- e 1)) ))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e )))
```

- The ... expression may have more than one case (as in problems **B**, **D**, and **G** of Lisp Assignment 4): The ... expression may, e.g., be a **COND** or **IF** expression.
-
-

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written as follows:

```
(defun f (e)
  (if (null e) or (zerop e)
      value of (f nil) or (f 0)
      (let ((X (f (cdr e)) or (f (- e 1)) ))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e )))
```

- The ... expression may have more than one case (as in problems **B**, **D**, and **G** of Lisp Assignment 4): The ... expression may, e.g., be a **COND** or **IF** expression.
- If there is no case in which **X** is used more than once, then ***eliminate the LET***.
-

- Recursive functions of one argument, which is a list or a nonnegative integer, can often be written as follows:

```
(defun f (e)
  (if (null e) or (zerop e)
      value of (f nil) or (f 0)
      (let ((X (f (cdr e)) or (f (- e 1))))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e )))
```

- The ... expression may have more than one case (as in problems **B**, **D**, and **G** of Lisp Assignment 4): The ... expression may, e.g., be a **COND** or **IF** expression.
- If there is no case in which **X** is used more than once, then ***eliminate the LET***.
- If the LET isn't eliminated, ***move any case in which X needn't be used out of the LET***. If the LET ***is*** eliminated but ***there's a case where the recursive call's result isn't needed, deal with such cases as base cases--i.e., without making a recursive call***.

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

Example Write a function **evens** such that:
If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil.**

-

-

-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- Note that the problem specification has this form:

  "***If*** *l ⇒ a proper list of integers*, ***then*** … "

  This means our function will ***not*** be obligated to do anything in particular when its argument value is ***not*** a proper list of integers:

-

-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil.**

- Note that the problem specification has this form:

  "***If*** *l ⇒ a proper list of integers*, ***then*** … "

  This means our function will ***not*** be obligated to do anything in particular when its argument value is ***not*** a proper list of integers: *It is logically impossible to violate the specification in that case!*

-

-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil.**

- Note that the problem specification has this form:

  "***If*** *l ⇒ a proper list of integers*, ***then*** … "

  This means our function will ***not*** be obligated to do anything in particular when its argument value is ***not*** a proper list of integers: *It is logically impossible to violate the specification in that case!*

- This is analogous to the meaning of a rule such as:

  If you enter this exhibit, then you must buy a ticket.

  This rule does ***not*** obligate you to do anything if you do ***not*** enter the exhibit: *It is logically impossible to violate this rule if you do not enter the exhibit.*

-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- Note that the problem specification has this form:

  "***If*** *l ⇒ a proper list of integers*, ***then*** … "

  This means our function will ***<u>not</u>*** be obligated to do anything in particular when its argument value is ***<u>not</u>*** a proper list of integers: *It is logically impossible to violate the specification in that case!*

- This is analogous to the meaning of a rule such as:

  If you enter this exhibit, then you must buy a ticket.

  This rule does ***<u>not</u>*** obligate you to do anything if you do ***<u>not</u>*** enter the exhibit: *It is logically impossible to violate this rule if you do not enter the exhibit.*

- If its argument value is ***<u>not</u>*** a proper list of integers, then our function ***evens*** *may return any value whatsoever or produce an evaluation error* without violating the specification!

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil.**

- If its argument value is ***<u>not</u>*** a proper list of integers, then our function ***evens*** *may return any value whatsoever or produce an evaluation error* without violating the specification!

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- If its argument value is ***not*** a proper list of integers, then our function *evens* may return any value whatsoever or produce an evaluation error without violating the specification!
-
-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- If its argument value is ***<u>not</u>*** a proper list of integers, then our function *evens* may return any value whatsoever or produce an evaluation error without violating the specification!
- The recursive functions you are asked to write will often be specified like this (i.e., with preconditions on argument values that the function may ***<u>assume</u>*** to be satisfied).
-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- If its argument value is ***<u>not</u>*** a proper list of integers, then our function *evens* may return any value whatsoever or produce an evaluation error without violating the specification!
- The recursive functions you are asked to write will often be specified like this (i.e., with preconditions on argument values that the function may ***<u>assume</u>*** to be satisfied).
- As a general rule, code that checks the validity of argument values should ***<u>not</u>*** be put into short recursive functions: Such checks would complicate/lengthen the code, and may be repeated unnecessarily at every recursive call.
  -
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- If its argument value is ***not*** a proper list of integers, then our function *evens* may return any value whatsoever or produce an evaluation error without violating the specification!
- The recursive functions you are asked to write will often be specified like this (i.e., with preconditions on argument values that the function may ***assume*** to be satisfied).
- As a general rule, code that checks the validity of argument values should ***not*** be put into short recursive functions: Such checks would complicate/lengthen the code, and may be repeated unnecessarily at every recursive call.
  - Such checks may be done in "gatekeeper" functions that are used by other code to call the recursive functions.
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- If its argument value is ***not*** a proper list of integers, then our function *evens* may return any value whatsoever or produce an evaluation error without violating the specification!
- The recursive functions you are asked to write will often be specified like this (i.e., with preconditions on argument values that the function may ***assume*** to be satisfied).
- As a general rule, code that checks the validity of argument values should ***not*** be put into short recursive functions: Such checks would complicate/lengthen the code, and may be repeated unnecessarily at every recursive call.
  - Such checks may be done in "gatekeeper" functions that are used by other code to call the recursive functions.
  - Assignments 4 & 5 don't ask you to write such "gatekeeper" functions, but only the recursive functions themselves!

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

•

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun f (e)
  (if (null e)
      value of (f nil)
      (let ((X (f (cdr e))))
        an expression that ⇒ value of (f e)
        and that involves X and, possibly, e
```

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      value of (evens nil)
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L  )))
```

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L )))
```

-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L  )))
```

- To write the ... expression, let's first consider *<u>one</u> possible value of* **L**, *the resulting value of* **X**, and *what* ... *'s value should be for that value of* **L**:

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        [an expression that ⇒ value of (evens L)
         and that involves X and, possibly, L] )))
```

- To write the […] expression, let's first consider *<u>one</u> possible value of* **L**, *the resulting value of* **X**, and *what* […]*'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**,

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L ))))
```

- To write the ... expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and *what* ... *'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ .
  Then **X** ⇒ and ... should ⇒ .

  ○

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L      )))
```

- To write the $\boxed{\ldots}$ expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and *what* $\boxed{\ldots}$ *'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ and $\boxed{\ldots}$ should ⇒ .

  ○

Example Write a function **evens** such that:
If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L    )))
```

- To write the ... expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and *what* ... *'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and ... should ⇒ .

  ○

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3)) ⇒ (2 4 0 2)**; **(evens nil) ⇒ nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L    )))
```

- To write the ... expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and *what* ... *'s value should be for that value of* **L**:

  Suppose **L ⇒ (7 2 -1 4 0 9 2 3)**, so **(cdr L) ⇒ (2 -1 4 0 9 2 3)**.
  Then **X ⇒ (2 4 0 2)** and ... should ⇒ **(2 4 0 2)**.

  ○

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L        )))
```

- To write the … expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and *what* … *'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and … should ⇒ **(2 4 0 2)**.

  - For ***this*** **L**, what is a good … expression?

Example Write a function **evens** such that:
If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

So **(evens '(7 2 -1 4 0 9 2 3))** ⇒ **(2 4 0 2)**; **(evens nil)** ⇒ **nil**.

- We'll solve this problem in the way that was described above:

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L    )))
```

- To write the $\boxed{\dots}$ expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and *what* $\boxed{\dots}$*'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and $\boxed{\dots}$ should ⇒ **(2 4 0 2)**.
  - For ***this*** **L**, what is a good $\boxed{\dots}$ expression? **Ans.: X**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L   )))
```

- To write the $\boxed{\dots}$ expression, let's first consider *<u>one</u> possible value of* **L**, *the resulting value of* **X**, and *what* $\boxed{\dots}$*'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and $\boxed{\dots}$ should ⇒ **(2 4 0 2)**.

  - For ***<u>this</u>*** **L**, what is a good $\boxed{\dots}$ expression? **Ans.: X**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L )))
```

- To write the $\boxed{\dots}$ expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and what $\boxed{\dots}$*'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and $\boxed{\dots}$ should ⇒ **(2 4 0 2)**.

  - For ***this*** **L**, what is a good $\boxed{\dots}$ expression? **Ans.: X**
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L  )))
```

- To write the … expression, let's first consider <u>*one*</u> *possible value of* **L**, *the resulting value of* **X**, and what …*'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and … should ⇒ **(2 4 0 2)**.

  - For ***<u>this</u>*** **L**, what is a good … expression? **Ans.: X**
  - Is **X** a good … for ***<u>all</u>*** non-null values of **L**? If not, ***<u>when</u>*** is **X** a good … ? **Ans.**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L   )))
```

- To write the … expression, let's first consider *one possible value of* **L**, *the resulting value of* **X**, and what … *'s value should be for that value of* **L**:

  Suppose **L** ⇒ **(7 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and … should ⇒ **(2 4 0 2)**.

  - For ***this*** **L**, what is a good … expression? **Ans.: X**
  - Is **X** a good … for ***all*** non-null values of **L**? If not, ***when*** is **X** a good … ? **Ans.** It's good if **(oddp (car L))**.

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L    )))
```

•

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then

(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L   )))
```

- We've seen that **X** is a good ... if (oddp (car L)). To find a good ... if (***not*** (oddp (car L))), we try ***<u>another example</u>***:

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L      )))
```

- We've seen that **X** is a good ... if (oddp (car L)). To find a good ... if (***not*** (oddp (car L))), we try ***<u>another example</u>***:

  Suppose **L** ⇒ (**4 2 -1 4 0 9 2 3**), so **(cdr L)** ⇒ .
  Then **X** ⇒ and ... should ⇒ .

  -
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then

(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L ))))
```

- We've seen that **X** is a good $\boxed{\ldots}$ if (oddp (car L)). To find a good $\boxed{\ldots}$ if (***not*** (oddp (car L))), we try ***<u>another example</u>***:

  Suppose **L** ⇒ **(4 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.

  Then **X** ⇒ and $\boxed{\ldots}$ should ⇒ .

  -
  -

Example Write a function **evens** such that:
If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L )))
```

- We've seen that **X** is a good […] if (oddp (car L)). To find a good […] if (***not*** (oddp (car L))), we try ***<u>another example</u>***:
  Suppose **L** ⇒ **(4 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and […] should ⇒ .
  -
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L )))
```

- We've seen that **X** is a good ... if (oddp (car L)). To find a good ... if (***not*** (oddp (car L))), we try ***<u>another example</u>***:
  Suppose **L** ⇒ **(4 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and ... should ⇒ **(4 2 4 0 2)**.
  -
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
         an expression that ⇒ value of (evens L)
         and that involves X and, possibly, L    )))
```

- We've seen that **X** is a good $\boxed{\dots}$ if (oddp (car L)). To find a good $\boxed{\dots}$ if (***not*** (oddp (car L))), we try ***<u>another example</u>***:

  Suppose **L** ⇒ **(4 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and $\boxed{\dots}$ should ⇒ **(4 2 4 0 2)**.

  - For ***<u>this</u>*** **L**, what is a good $\boxed{\dots}$ expression?
    Ans.:
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L )))
```

- We've seen that X is a good ... if (oddp (car L)). To find a good ... if (***not*** (oddp (car L))), we try ***another example***:

  Suppose **L** ⇒ **(4 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and ... should ⇒ **(4 2 4 0 2)**.
  - For ***this*** **L**, what is a good ... expression?
    Ans.: (cons (car L) **X**).
  -

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by ***omitting*** *its odd elements.*

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L    )))
```

- We've seen that X is a good $\boxed{\ldots}$ if (oddp (car L)). To find a good $\boxed{\ldots}$ if (***not*** (oddp (car L))), we try ***another example***:

  Suppose L ⇒ (4 2 -1 4 0 9 2 3), so (cdr L) ⇒ (2 -1 4 0 9 2 3).
  Then X ⇒ (2 4 0 2) and $\boxed{\ldots}$ should ⇒ (4 2 4 0 2).

  - For ***this*** L, what is a good $\boxed{\ldots}$ expression?
    Ans.: (cons (car L) X).
  - Is (cons (car L) X) a good $\boxed{\ldots}$ expression for ***all*** values of L such that (***not*** (oddp (car L)))? **Ans.**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L     )))
```

- We've seen that X is a good ... if (oddp (car L)). To find a good ... if (*not* (oddp (car L))), we try ***<u>another example</u>***:

  Suppose L ⇒ (4 2 -1 4 0 9 2 3), so (cdr L) ⇒ (2 -1 4 0 9 2 3).
  Then X ⇒ (2 4 0 2) and ... should ⇒ (4 2 4 0 2).

  - For ***<u>this</u>*** L, what is a good ... expression?
    Ans.: (cons (car L) X).
  - Is (cons (car L) X) a good ... expression for ***<u>all</u>*** values of L such that (*not* (oddp (car L)))? **Ans. YES!**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then

(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L )))
```

- We've seen that X is a good ... if (oddp (car L)).
-
-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L  )))
```

- We've seen that X is a good $\boxed{\cdots}$ if (oddp (car L)).
- We've seen that (cons (car L) X) is a good $\boxed{\cdots}$ if (*not* (oddp (car L))).
-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        an expression that ⇒ value of (evens L)
        and that involves X and, possibly, L     )))
```

- We've seen that X is a good ... if (oddp (car L)).
- We've seen that (cons (car L) X) is a good ... if (*not* (oddp (car L))).
- So now we can write ... as:

```
(cond ((oddp (car L)) X)
      (t (cons (car L) X)))
```

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X)))))))
```

- We've seen that **X** is a good $\boxed{\dots}$ if (oddp (car L)).
- We've seen that (cons (car L) **X**) is a good $\boxed{\dots}$ if (***not*** (oddp (car L))).
- So now we can write $\boxed{\dots}$ as:

```
(cond ((oddp (car L)) X)
      (t (cons (car L) X)))
```

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then

(evens l) ⇒ a list obtained from l by omitting its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X))))))
```

- We've seen that **X** is a good ... if (oddp (car L)).
- We've seen that (cons (car L) **X**) is a good ... if (*not* (oddp (car L))).
- So now we can write ... as shown above!

Q.

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by __omitting__ its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X))))))
```

- We've seen that X is a good ... if (oddp (car L)).
- We've seen that (cons (car L) X) is a good ... if (not (oddp (car L))).
- So now we can write ... as shown above!

Q. Is there any case in which **X** is used ***more than once***?

A.

Example Write a function **evens** such that:

If L ⇒ a proper list of integers, then
(evens L) ⇒ a list obtained from L by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X)))))))
```

- We've seen that **X** is a good $\boxed{\cdots}$ if (oddp (car L)).
- We've seen that (cons (car L) **X**) is a good $\boxed{\cdots}$ if (***not*** (oddp (car L))).
- So now we can write $\boxed{\cdots}$ as shown above!

Q. Is there any case in which **X** is used ***<u>more than once</u>***?

A. No! X is used ***<u>just once</u> in each of the 2 cases*** of the **cond.**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X))))))
```

Q. Is there any case in which **X** is used ***more than once***?

A. No! X is used ***just once in each of the 2 cases*** of the **cond.**

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X))))))
```

Q. Is there any case in which **X** is used ***<u>more than once</u>***?

A. No! X is used ***<u>just once</u> in each of the 2 cases*** of the **cond.**

•

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

```
(defun evens (L)
  (if (null L)
      nil
      (let ((X (evens (cdr L))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X)))))))
```

Q. Is there any case in which **X** is used ***<u>more than once</u>***?

A. No! X is used ***<u>just once</u> in each of the 2 cases*** of the **cond.**

- So we can ***<u>eliminate the LET</u>*** and substitute **(evens (cdr L))** for each occurrence of **X**, to simplify the definition.

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

(defun evens (L)
  (if (null L)
      nil
      ~~(let ((X (evens (cdr L))))~~
        (cond ((oddp (car L)) (evens (cdr L)) ~~X~~)
              (t (cons (car L) (evens (cdr L)) ~~X~~)))~~)~~))

Q. Is there any case in which **X** is used ***more than once***?

A. No! X is used ***just once in each of the 2 cases*** of the **cond.**

- So we have *eliminated the LET* and substituted **(evens (cdr L))** for each occurrence of **X**, to simplify the definition.

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

(defun evens (L)
(if (null L)
nil
~~(let ((X (evens (cdr L))))~~
(cond ((oddp (car L)) **(evens (cdr L))** ~~X~~)
(t (cons (car L) **(evens (cdr L))** ~~X~~))) ~~)~~))

- We have ***eliminated the LET*** and substituted **(evens (cdr L))** for each occurrence of **X**, to simplify the definition.

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by <u>omitting</u> its odd elements.

(defun evens (L)
(if (null L)
nil
~~(let ((X (evens (cdr L))))~~
(cond ((oddp (car L)) **(evens (cdr L))** ~~X~~)
(t (cons (car L) **(evens (cdr L))** ~~X~~)))~~)~~**))**

- We have ***eliminated the LET*** and substituted **(evens (cdr L))** for each occurrence of **X**, to simplify the definition.
-

Example Write a function **evens** such that:

If l ⇒ a proper list of integers, then
(evens l) ⇒ a list obtained from l by omitting its odd elements.

(defun evens (L)
(if (null L)
nil
~~(let ((X (evens (cdr L))))~~
(cond ((oddp (car L)) **(evens (cdr L))** ~~X~~)
(t (cons (car L) **(evens (cdr L))** ~~X~~)))~~)~~**))**

- We have ***eliminated the LET*** and substituted **(evens (cdr L))** for each occurrence of **X**, to simplify the definition.
- To further simplify the definition, we can replace (**if** (null L) nil (**cond** …)) with (**cond** ((null L) nil) …):

```
(defun evens (L)
  (cond ((null L) nil)
        ((oddp (car L)) (evens (cdr L)))
        (t (cons (car L) (evens (cdr L))))))
```