

Another example, from p. 117 of Touretzky:

```
(defun where-is (x)
  (cond ((equal x 'paris) 'france)
        ((equal x 'london) 'england)
        ((equal x 'beijing) 'china)
        (t 'unknown)))
```

From pp. 119 – 20 of Touretzky:

Parenthesis errors can play havoc with COND expressions. Most COND clauses begin with **exactly two parentheses**. The first marks the beginning of the clause, and the second marks the beginning of the clause's test.

...

If the test part of a clause is just a symbol, not a call to a function, then the clause should begin with **a single parenthesis**. Notice that in WHERE-IS the clause with T as the test begins with only one parenthesis.

From p. 120 of Touretzky:

Here are two of the more common parenthesis errors made with COND. First, suppose we leave a parenthesis out of a COND clause. What would happen?

```
(cond (equal x 'paris 'france)
      (. . .)
      (. . .)
      (t 'unknown))
```

The first clause of the COND starts with only one left parenthesis instead of two. As a result, the test part of this clause is just the symbol EQUAL. When the test is evaluated, EQUAL will cause an unassigned variable error.

On the other hand, consider what happens when too many parentheses are used:

```
(cond ((. . .) 'france)
      ((. . .) 'england)
      ((. . .) 'china)
      ((t 'unknown)))
```

If X has the value HACKENSACK, we will reach the fourth COND clause. Due to the presence of an extra pair of parentheses in this clause, the test is (T 'UNKNOWN) instead of simply T. T is not a function, so this test will generate an undefined function error.

The OR Macro Operator

$(\text{OR } e_1 \dots e_n)$ is analogous to $e_1 \parallel \dots \parallel e_n$ in C++ or Java, except that when $(\text{OR } e_1 \dots e_n)$'s value is *true* its value is *the value of the first e whose value isn't NIL*.

From p. 122 of Touretzky:

suppose we want a function GTEST that takes two numbers as input and returns T if either the first is greater than the second or one of them is zero. These conditions form a disjunctive set; only one need be true for GTEST to return T. OR is used for disjunctions.

```
(defun gtest (x y)
  (or (> x y)
      (zerop x)
      (zerop y)))
```

The OR Macro Operator

$(\text{OR } e_1 \dots e_n)$ is analogous to $e_1 \parallel \dots \parallel e_n$ in C++ or Java, except that when $(\text{OR } e_1 \dots e_n)$'s value is *true* its value is the value of the first e whose value isn't NIL.

Another Example Suppose f is defined as follows:

```
(defun f (x) (or (member x '(A B C)) (member x '(2 3 B))))
```

Then:

- $(f \text{ 'B}) \Rightarrow (B \ C)$
- $(f \text{ 'A}) \Rightarrow (A \ B \ C)$
- $(f \ 2) \Rightarrow (2 \ 3 \ B)$
- $(f \ 6) \Rightarrow \text{NIL}$

Like $e_1 \parallel \dots \parallel e_n$ in C++ or Java, $(\text{OR } e_1 \dots e_n)$ is evaluated using short-circuit evaluation, as follows:

- The expressions e_1, \dots, e_n are evaluated in that order, but evaluation of these expressions stops when an expression e_i is found to have a value that isn't NIL: When that happens, the value of e_i is returned as the value of $(\text{OR } e_1 \dots e_n)$, and any subsequent expressions e_{i+1}, \dots, e_n are not evaluated.
- If e_1, \dots, e_n all have value NIL, then the value of $(\text{OR } e_1 \dots e_n)$ is also NIL.

Like $e_1 \parallel \dots \parallel e_n$ in C++ or Java, $(\text{OR } e_1 \dots e_n)$ is evaluated using *short-circuit evaluation*, as follows:

- The expressions e_1, \dots, e_n are evaluated in that order, but evaluation of these expressions stops when an expression e_i is found to have a value that isn't NIL: When that happens, the value of e_i is returned as the value of $(\text{OR } e_1 \dots e_n)$, and any subsequent expressions e_{i+1}, \dots, e_n are not evaluated.
- If e_1, \dots, e_n all have value NIL, the value of $(\text{OR } e_1 \dots e_n)$ is NIL.

From p. 123 of Touretzky:

`(or 'fee 'fie 'foe)` \Rightarrow `fee` (more precisely, `FEE`)

`(or nil 'foe nil)` \Rightarrow `foe`

`(or nil t t)` \Rightarrow `t`

`(or 'george nil 'harry)` \Rightarrow `george`

`(or 'george 'fred 'harry)` \Rightarrow `george`

`(or nil 'fred 'harry)` \Rightarrow `fred`

The AND Macro Operator

$(\text{AND } e_1 \dots e_n)$ is analogous to $e_1 \ \&\& \dots \&\& e_n$ in C++ or Java, except that when $(\text{AND } e_1 \dots e_n)$'s value is *true* its value is the value of e_n (which will not be NIL but need not be T).

From p. 122 of Touretzky:

Suppose we want a predicate for small (no more than two digit) positive odd numbers. We can use AND to express this conjunction of simple conditions:

```
(defun small-positive-oddp (x)
  (and (< x 100)
       (> x 0)
       (oddp x)))
```

The AND Macro Operator

$(\text{AND } e_1 \dots e_n)$ is analogous to $e_1 \ \&\& \dots \&\& e_n$ in C++ or Java, except that when $(\text{AND } e_1 \dots e_n)$'s value is *true* its value is the value of e_n (which will not be NIL but need not be T).

Another Example Suppose g is defined as follows:

```
(defun g (x) (and (member x '(A B)) (member x '(C B 2 3))))
```

Then:

- $(g \ 'B) \Rightarrow (B \ 2 \ 3)$
- $(g \ 'A) \Rightarrow \text{NIL}$
- $(g \ 2) \Rightarrow \text{NIL}$
- $(g \ 6) \Rightarrow \text{NIL}$

Like $e_1 \ \&\& \dots \&\& e_n$ in C++ or Java, $(\text{AND } e_1 \dots e_n)$ is evaluated using **short-circuit evaluation**, as follows:

- The expressions e_1, \dots, e_n are evaluated in that order, but evaluation of these expressions stops when an expression e_i is found to have value NIL: When that happens, NIL is returned as the value of $(\text{AND } e_1 \dots e_n)$, and any subsequent expressions e_{i+1}, \dots, e_n are not evaluated.
- If e_1, \dots, e_n all have non-NIL values, then the value of $(\text{AND } e_1 \dots e_n)$ is **the value of e_n** .

Like $e_1 \ \&\& \dots \&\& \ e_n$ in C++ or Java, $(\text{AND } e_1 \dots e_n)$ is evaluated using *short-circuit evaluation*, as follows:

- The expressions e_1, \dots, e_n are evaluated in that order, but *evaluation of these expressions stops when an expression e_i is found to have value NIL*: When that happens, NIL is returned as the value of $(\text{AND } e_1 \dots e_n)$; any subsequent expressions e_{i+1}, \dots, e_n are not evaluated.
- If e_1, \dots, e_n all have non-NIL values, then the value of $(\text{AND } e_1 \dots e_n)$ is *the value of e_n* .

From p. 123 of Touretzky:

`(and 'fee 'fie 'foe) ⇒ foe` (more precisely, FOE)

`(and 'fee 'fie nil) ⇒ nil`

`(and (equal 'abc 'abc) 'yes) ⇒ yes`

`(and 'george nil 'harry) ⇒ nil`

`(and 'george 'fred 'harry) ⇒ harry`

`(and 1 2 3 4 5) ⇒ 5`

LET and LET*

LET gives values to *local* variables for use in an expression.

Example: `(let ((x (- 2 1))
 (y 3)
 (z (* 2 4)))
 (+ x (* y z)))` \Rightarrow **25**

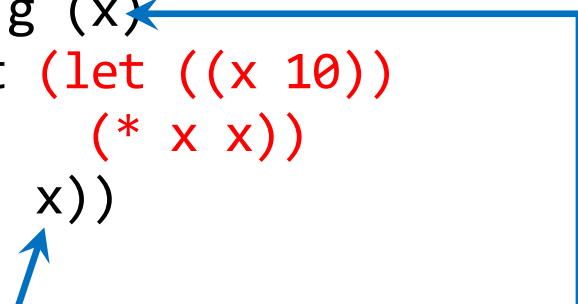
This LET expression can be understood as meaning

`(+ x (* y z))` **where** `x = (- 2 1)`, `y = 3`, `z = (* 2 4)`

The scope of the local variables introduced by a LET expression is confined to the body of the expression.

- To illustrate this, suppose we define a function as follows:

```
(defun g (x)  
  (list (let ((x 10))  
          (* x x))  
        x))
```



Then this `x` is the parameter `x` of `g`, which is unrelated to the local variable `x` of the LET!

Hence: `(g 3)` \Rightarrow **(100 3)**

Examples from pp. 141 – 3 of Touretzky:

So far, the only local variables we've seen have been those created by calling user-defined functions, such as `DOUBLE` or `AVERAGE`. Another way to create a local variable is with the `LET` special function. For example, since the average of two numbers is half their sum, we might want to use a local variable called `SUM` inside our `AVERAGE` function. We can use `LET` to create this local variable and give it the desired initial value. Then, in the body of the `LET` form, we can compute the average.

```
(defun average (x y)
  (let ((sum (+ x y)))
    (list x y 'average 'is (/ sum 2.0)))))
```

```
> (average 3 7)
(3 7 AVERAGE IS 5.0)
```

The right way to read a `LET` form such as

```
(let ((x 2)
      (y 'aardvark))
  (list x y))
```

is to say “Let `X` be 2, and `Y` be `AARDVARK`; return `(LIST X Y)`.”

Examples from pp. 141 – 3 of Touretzky:

```
(defun average (x y)
  (let ((sum (+ x y)))
    (list x y 'average 'is (/ sum 2.0)))))
```

```
> (average 3 7)
(3 7 AVERAGE IS 5.0)
```

```
(defun switch-billing (x)
  (let ((star (first x))
        (co-star (third x)))
    (list co-star 'accompanied 'by star)))
```

```
> (switch-billing '(fred and ginger))
(GINGER ACCOMPANIED BY FRED)
```

- These examples illustrate one reason we use LET (or LET*): *To give meaningful names (e.g., sum, star, and co-star) to the values of certain expressions and so make code more readable.* Another reason to use LET or LET* will be discussed later.

Evaluation of LET Forms

The

general syntax of LET is:

```
(LET ((var-1 value-1)
      (var-2 value-2)
      ...
      (var-n value-n))
  body)
```

This is from p. 142 of Touretzky.

In functional programming, body consists of just one expression whose value will be returned as the value of the entire LET form.

The first argument to LET is a list of variable-value pairs. The n value forms are evaluated, **then** n local variables are created to hold the results, **finally** the forms in the body of the LET are evaluated.

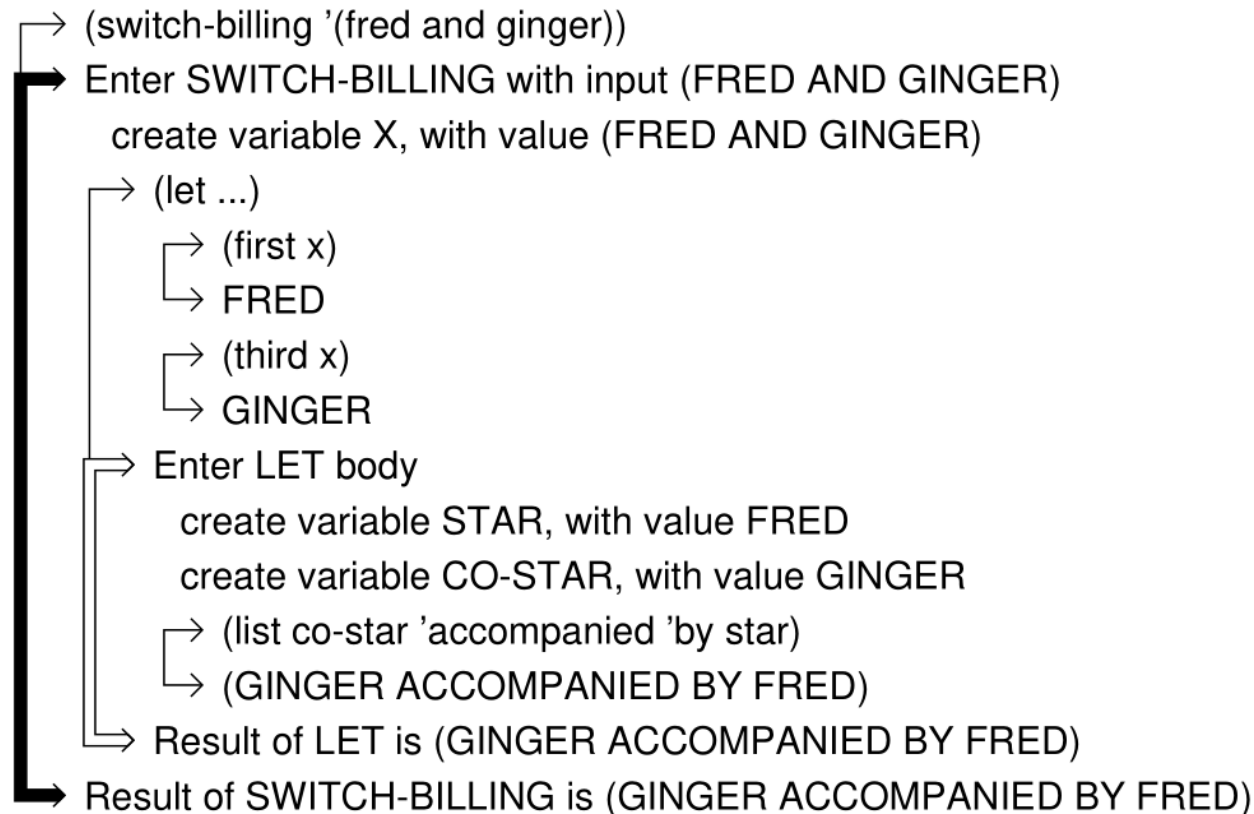
- The expressions `value-1`, ... , `value-n` are evaluated using the variable bindings that existed just before the LET expression.
- The local variables `var-1`, ... , `var-n` will be given the values of the expressions `value-1`, ... , `value-n`, but they are **not** visible when those expressions are being evaluated!
- Once the `body` expression has been evaluated, the n local variables cease to exist: `var-1`, ... , `var-n` will then have the values, if any, that they had before the LET expression.

From p. 143 of
Touretzky.

```
(defun switch-billing (x)
  (let ((star (first x))
        (co-star (third x)))
    (list co-star 'accompanied 'by star)))
```

```
> (switch-billing '(fred and ginger))
(GINGER ACCOMPANIED BY FRED)
```

Here is an evaltrace showing exactly how LET creates the local variables STAR and CO-STAR. Note that the two value forms, (FIRST X) and (THIRD X), are both evaluated before any local variables are created.



Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
            (b (* x 2))
            (c (+ x y)))
      (/ (+ a b c) w))
    x))
```

Evaluation of (g 3 4 7):

```
w ⇒ 3    x ⇒ 4    y ⇒ 7
      (sqrt x) ⇒ 2
      (* x 2) ⇒ 8
      (+ x y) ⇒ 11
```

LET \Rightarrow

X \Rightarrow

$(g \ 3 \ 4 \ 7) \Rightarrow$

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
      (/ (+ a b c) w))
    x))
```

Evaluation of (g 3 4 7):

w \Rightarrow 3 x \Rightarrow 4 y \Rightarrow 7

the LET's local a \Rightarrow 2

the LET's local b \Rightarrow 8

the LET's local c \Rightarrow 11

LET \Rightarrow

x \Rightarrow

(g 3 4 7) \Rightarrow

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
      (/ (+ a b c) w))
    x))
```

Evaluation of (g 3 4 7):

w \Rightarrow 3 x \Rightarrow 4 y \Rightarrow 7

the LET's local a \Rightarrow 2

the LET's local b \Rightarrow 8

the LET's local c \Rightarrow 11

LET \Rightarrow (2+8+11)/3 = 7

x \Rightarrow 4

(g 3 4 7) \Rightarrow 7+4 = 11

A related example:

```
(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
      (sqrt x) ⇒ 2
      (* x 2) ⇒ 8
      (+ x y) ⇒ 11

```

LET \Rightarrow

X \Rightarrow

(h 3 4 7) \Rightarrow

A related example:

```
(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 \Rightarrow 3 x \Rightarrow 4 y \Rightarrow 7

the LET's local x \Rightarrow 2

the LET's local y \Rightarrow 8

the LET's local z \Rightarrow 11

LET \Rightarrow

x \Rightarrow

(h 3 4 7) \Rightarrow

Another example:

```
(defun g (w x y)
  (+ (let ((a (sqrt x))
           (b (* x 2))
           (c (+ x y)))
      (/ (+ a b c) w))
    x))
```

Evaluation of (g 3 4 7):

w \Rightarrow 3 x \Rightarrow 4 y \Rightarrow 7
the LET's local a \Rightarrow 2
the LET's local b \Rightarrow 8
the LET's local c \Rightarrow 11
LET \Rightarrow (2+8+11)/3 = 7
x \Rightarrow 4

(g 3 4 7) \Rightarrow 7+4 = 11

A related example:

```
(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 \Rightarrow 3 x \Rightarrow 4 y \Rightarrow 7
the LET's local x \Rightarrow 2
the LET's local y \Rightarrow 8
the LET's local z \Rightarrow 11
LET \Rightarrow (2+8+11)/3 = 7
x \Rightarrow 4

(h 3 4 7) \Rightarrow 7+4 = 11

Evaluation of

```
(let ((x1 expr1)  
      ⋮  
      (xn exprn))  
  body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x1 ... xn) body)
```

and then calling the function as follows:

```
(my-helper-function expr1 ... exprn)
```

For example, evaluation of

```
(let ((a (sqrt x))  
      (b (* x 2))  
      (c (+ x y)))  
  (/ (+ a b c) 5))
```

is roughly equivalent to making a definition

```
(defun my-helper-function (a b c) (/ (+ a b c) 5))
```

and then calling the function as follows:

```
(my-helper-function (sqrt x) (* x 2) (+ x y))
```

Evaluation of

```
(let ((x1 expr1)  
      ⋮  
      (xn exprn))  
  body)
```

is roughly equivalent to making a definition

```
(defun my-helper-function (x1 ... xn) body)
```

and then calling the function as follows:

```
(my-helper-function expr1 ... exprn)
```

In fact

```
(let ((x1 expr1)  
      ⋮  
      (xn exprn))  
  body)
```

is essentially equivalent to:

```
((lambda (x1 ... xn) body) expr1 ... exprn)
```

- The latter expression calls a function that's the same as `my-helper-function` but has not been given a name.

LET*

LET* forms are equivalent to nested LET forms:

$$\begin{array}{ccc} (\text{let}^* ((x_1 \text{ expr}_1) & & (\text{let} ((x_1 \text{ expr}_1)) \\ & (x_2 \text{ expr}_2) & = & (\text{let} ((x_2 \text{ expr}_2)) \\ & \vdots & & \vdots \\ & (x_n \text{ expr}_n)) & & (\text{let} ((x_n \text{ expr}_n)) \\ \text{body}) & & \text{body) ... }) \end{array}$$

- Thus for $2 \leq k \leq n$ each expression expr_k *can use the previous local variables* x_1, \dots, x_{k-1} (which would not be the case if we replaced LET* with LET).

On p. 144 of Touretzky, the difference between LET* and LET is described as follows:

The LET* special function is similar to LET, except it creates the local variables one at a time instead of all at once. Therefore, the first local variable forms part of the lexical context in which the value of the second variable is computed, and so on. This way of creating local variables is useful when one wants to assign names to several intermediate steps in a long computation.

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)))
```

```
> (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, \dots, 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

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

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))
```

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 \Rightarrow 3 x \Rightarrow 4 y \Rightarrow 7
the LET's local x \Rightarrow 2
the LET's local y \Rightarrow 8
the LET's local z \Rightarrow 11
LET \Rightarrow (2+8+11)/3 = 7
x \Rightarrow 4

(h 3 4 7) \Rightarrow 7+4 = 11

Evaluation of (h* 3 4 7):

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

(h* 3 4 7) \Rightarrow 4+4 = 8

Recursive Functions

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 $l \Rightarrow$ a proper list, then
(length-of l) \Rightarrow 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 $L \Rightarrow$ a proper list, then $(\text{length-of } L) \Rightarrow$ the length of L .

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

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

- If $L \Rightarrow \text{NIL}$, 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 $L \Rightarrow$ a proper list, then $(\text{length-of } L) \Rightarrow$ the length of L .

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

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

To make our function good even when $L \Rightarrow \text{NIL}$, 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 $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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 $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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 $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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 $(\text{length-of } (\text{cdr } L))$ returns the right result, **then** the call $(\text{length-of } L)$ returns the right result.
- So, for all $n > 0$, **if** $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length $< n$, **then** $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length n .
- **Example:** **If** $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length 0, 1, 2, or 3, **then** $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length 4.

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

If $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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* $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length $< n$, *then* $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length n .
 - **Example:** *If* $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length 0, 1, 2, or 3, *then* $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length 4.
 - $(\text{length-of } l)$ returns the right result (i.e., 0) when $l \Rightarrow$ a list of length 0.
- \therefore If $l \Rightarrow$ a proper list of any length,
then $(\text{length-of } l) \Rightarrow$ the right result (i.e., l 's length).

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

If $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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 $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length $< n$, then $(\text{length-of } l)$ returns the right result when $l \Rightarrow$ a proper list of length n .
- $(\text{length-of } l)$ returns the right result (i.e., 0) when $l \Rightarrow$ a list of length 0.

\therefore If $l \Rightarrow$ a proper list of any length,
then $(\text{length-of } l) \Rightarrow$ 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 $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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 $l \Rightarrow$ a proper list, then $(\text{length-of } l) \Rightarrow$ 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 $(\text{length-of } (\text{cdr } L))$:

```
(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!