**Example** Without using append, write a function **append-2** such that:

*If* L1 ⇒ a proper list *and* L2 ⇒ a proper list, *then*
(append-2 L1 L2) ⇒ a list that is equal to (append L1 L2)

**Final version:**

```
(defun append-2 (L1 L2)
  (if (null L1)
      L2
      (cons (car L1) (append-2 (cdr L1) L2))))
```

- In our definition of **append-2**, the ***first*** argument of its recursive call has a smaller value than the first argument of the current call, while the ***second*** argument has the same value in the recursive call as in the current call.
- Why can't we define **append-2** in the opposite way—i.e., by letting the ***second*** argument of its recursive call have a smaller value than the second argument of the current call, and letting the ***first*** argument have the same value in the recursive call as in the current call?

**Example** Without using append, write a function **append-2** such that:
*If* L1 ⇒ a proper list *and* L2 ⇒ a proper list, *then*
(append-2 L1 L2) ⇒ a list that is equal to (append L1 L2)

- Why can't we define **append-2** in the opposite way—i.e., by letting the ***second*** argument of its recursive call have a smaller value than the second argument of the current call, and letting the ***first*** argument have the same value in the recursive call as in the current call?

```
(defun append-2 (L1 L2)
  (if (null L2)                                   What will this be?
      value of (append-2 L1 nil)
      (let ((X (append-2 L1 (cdr L2))))
        an expression that ⇒ value of (append-2 L1 L2)
        and that involves X and, possibly, L1 and/or L2 )))
```

**Example** Without using append, write a function **append-2** such that:
*If* L1 ⇒ a proper list *and* L2 ⇒ a proper list, *then*
(append-2 L1 L2) ⇒ a list that is equal to (append L1 L2)

- Why can't we define **append-2** in the opposite way—i.e., by letting the ***second*** argument of its recursive call have a smaller value than the second argument of the current call, and letting the ***first*** argument have the same value in the recursive call as in the current call?

```
(defun append-2 (L1 L2)
  (if (null L2)                    Value of (append-2 L1 nil).
      L1
      (let ((X (append-2 L1 (cdr L2))))
        an expression that ⇒ value of (append-2 L1 L2)
        and that involves X and, possibly, L1 and/or L2 )))
```

- To consider how to write the $\boxed{\dots}$ expression, let's look at a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and what $\boxed{\dots}$*'s value should be in this case:*

**Example** Without using append, write a function **append-2** such that:

*If* L1 ⇒ a proper list *and* L2 ⇒ a proper list, *then*
(append-2 L1 L2) ⇒ a list that is equal to (append L1 L2)

```
(defun append-2 (L1 L2)
  (if (null L2)
      L1
      (let ((X (append-2 L1 (cdr L2))))
        an expression that ⇒ value of (append-2 L1 L2)
        and that involves X and, possibly, L1 and/or L2 )))
```

- To consider how to write the … expression, let's look at a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and *what* … *'s value should be in this case:*
- Suppose **L1** ⇒ **(1 2 3 4)** and **L2** ⇒ **(A B C D E)**,
  so **(cdr L2)** ⇒ **(B C D E)** and **X** ⇒ **(1 2 3 4 B C D E)**.
  For this L1 and L2, … should ⇒ **(1 2 3 4 A B C D E)**.
-

**Example** Without using append, write a function **append-2** such that:

*If* L1 ⇒ a proper list *and* L2 ⇒ a proper list, *then*
(append-2 L1 L2) ⇒ a list that is equal to (append L1 L2)

```
(defun append-2 (L1 L2)
  (if (null L2)
      L1
      (let ((X (append-2 L1 (cdr L2))))
        an expression that ⇒ value of (append-2 L1 L2)
        and that involves X and, possibly, L1 and/or L2 )))
```

- To consider how to write the … expression, let's look at a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and *what* … *'s value should be in this case:*
- Suppose **L1** ⇒ **(1 2 3 4)** and **L2** ⇒ **(A B C D E)**,
  so (cdr L2) ⇒ (B C D E) and **X** ⇒ **(1 2 3 4 B C D E)**.
  For this L1 and L2, … should ⇒ **(1 2 3 4 A B C D E)**.
- … would need to perform $\Theta$(length of **L1**) operations to create (append-2 L1 L2) from **L1**, **L2**, and **X** = (append-2 L1 (cdr L2)).

**Example** Without using append, write a function **append-2** such that:
*If* L1 ⇒ a proper list *and* L2 ⇒ a proper list, *then*
(append-2 L1 L2) ⇒ a list that is equal to (append L1 L2)

```
(defun append-2 (L1 L2)
  (if (null L2)
      L1
      (let ((X (append-2 L1 (cdr L2))))
        an expression that ⇒ value of (append-2 L1 L2)
        and that involves X and, possibly, L1 and/or L2 )))
```

- Suppose **L1** ⇒ **(1 2 3 4)** and **L2** ⇒ **(A B C D E)**,
so (cdr L2) ⇒ (B C D E) and **X** ⇒ **(1 2 3 4 B C D E)**.
For this L1 and L2, … should ⇒ **(1 2 3 4 A B C D E)**.
- … would need to perform Θ(length of **L1**) operations to create (append-2 L1 L2) from **L1**, **L2**, and **X** = (append-2 L1 **(cdr L2)**).
- So our original decision to have the ***first*** (rather than the second) argument of append-2 be smaller in the recursive call than the current call was right: That recursive strategy required each recursive call to perform only Θ(1) operations.

**Example** Write a function **all-numbers** such that:

*If L ⇒ a proper list, then*
*(all-numbers L) ⇒ T    if* ***every*** *element of the list is a number*
*(all-numbers L) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

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

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

**Example** Write a function **all-numbers** such that:

*If L ⇒ a proper list, then*
*(all-numbers L) ⇒ T    if every element of the list is a number*
*(all-numbers L) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

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

```
(defun all-numbers (L)
  (if (null L)
      T
      (let ((X (all-numbers (cdr L))))
        an expression that ⇒ value of (all-numbers L)
        and that involves X and, possibly, L         )))
```

We see from the spec that **(all-numbers nil) ⇒ T.**

-

**Example** Write a function **all-numbers** such that:

*If l ⇒ a proper list, then*
*(all-numbers l) ⇒ T if every element of the list is a number*
*(all-numbers l) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

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

```
(defun all-numbers (L)
  (if (null L)

        T

        (let ((X (all-numbers (cdr L))))
          an expression that ⇒ value of (all-numbers L)
          and that involves X and, possibly, L           )))
```

- We also see from the spec that **(and X (numberp (car L)))** would be a correct … expression, so we can now complete the definition!

**Example** Write a function **all-numbers** such that:

*If l ⇒ a proper list, then*
*(all-numbers l) ⇒ T if every element of the list is a number*
*(all-numbers l) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

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

```
(defun all-numbers (L)
  (if (null L)

      T

      (let ((X (all-numbers (cdr L))))
        (and X (numberp (car L))) )))
```

-

**Example** Write a function **all-numbers** such that:

*If L ⇒ a proper list, then*
*(all-numbers L) ⇒ T   if <u>every</u> element of the list is a number*
*(all-numbers L) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

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

```
 (defun all-numbers (L)
   (if (null L)

       T

       (let ((X (all-numbers (cdr L))))
         (and X (numberp (car L))) )))
```

- X is never used more than once, so we ***<u>eliminate the</u>* <u>LET</u>**:

```
(defun all-numbers (L)
  (if (null L)
      T
     ~~(let ((X (all-numbers (cdr L))))~~
        (and ~~X~~ (all-numbers (cdr L)) (numberp (car L)))~~)~~))
```

**Example** Write a function **all-numbers** such that:

*If l ⇒ a proper list, then*
*(all-numbers l) ⇒ T   if every element of the list is a number*
*(all-numbers l) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

- **X** is never used more than once, so we ***eliminate the LET***:

```
(defun all-numbers (L)
  (if (null L)
      T
      (and (all-numbers (cdr L)) (numberp (car L)))))
```

**RECALL**:

- 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***.

In the case **(numberp (car L)) ⇒ NIL**, the result of the recursive call **(all-numbers (cdr L))** ***isn't needed***, as the function will return NIL regardless of what that call returns!
***So we rewrite the code to deal with that case without the call.***

**Example** Write a function **all-numbers** such that:

*If l ⇒ a proper list, then*

*(all-numbers l) ⇒ T  if <u>every</u> element of the list is a number*

*(all-numbers l) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

- **X** is never used more than once, so we ***<u>eliminate the LET</u>***:

```
(defun all-numbers (L)
  (if (null L)
      T
      (and (numberp (car L)) (all-numbers (cdr L)))))
```

**<u>RECALL</u>**:

- If the LET isn't eliminated, ***<u>move any case in which X needn't be used out of the</u>*** <u>LET</u>. If the LET ***<u>is</u>*** eliminated but ***<u>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</u>***.

In the case **(numberp (car L)) ⇒ NIL**, the result of the recursive call **(all-numbers (cdr L))** ***<u>isn't needed</u>***, as the function will return NIL regardless of what that call returns!

***We've rewritten the code to deal with that case <u>without</u> the call.***

**Example** Write a function **all-numbers** such that:

*If l ⇒ a proper list, then*

*(all-numbers l) ⇒ T    if every element of the list is a number*

*(all-numbers l) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

```
(defun all-numbers (L)
  (if (null L)
      T
      (and (numberp (car L)) (all-numbers (cdr L)))))
```

- **Final cleanup:**

  Since **(if *c* T *e*)** = **(or *c* *e*)** if the value of *c* is always either T or NIL, we can simplify the above definition to:

```
(defun all-numbers (L)
  (or (null L)
      (and (numberp (car L)) (all-numbers (cdr L)))))
```

**Example** Write a function **all-numbers** such that:

*If l ⇒ a proper list, then*

*(all-numbers l) ⇒ T   if every element of the list is a number*

*(all-numbers l) ⇒ NIL otherwise.*

So: **(all-numbers '(6 2 6)) ⇒ T; (all-numbers '(7 1 DOG 9)) ⇒ NIL**

- **Final cleanup:**

  Since **(if *c* T *e*)** = **(or *c e*)** if the value of *c* is always either T or NIL, we can simplify the above definition to:

```
(defun all-numbers (L)
  (or (null L)
      (and (numberp (car L)) (all-numbers (cdr L)))))
```

**Common Mistake:**

The following will *not* work:

```
(defun all-numbers (L)
  (and (numberp (car L)) (all-numbers (cdr L))))
```

This returns NIL whenever the argument is a proper list!

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

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

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (let ((X (safe-sum (cdr L))))
        an expression that ⇒ value of (safe-sum L)
        and that involves X and, possibly, L          )))
```

We see from the spec that **(safe-sum nil) ⇒ 0.**

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are* ***not*** *all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (let ((X (safe-sum (cdr L))))
        an expression that ⇒ value of (safe-sum L)
        and that involves X and, possibly, L        )))
```

- To write the … expression, let's first consider *a possible value of* **L**, *the resulting value of* **X**, and what … *'s value should be for that value of* **L**:
- Suppose **L** ⇒ (7 2 4 9 3), so **(cdr L)** ⇒ **(2 4 9 3)**.
  Then **X** ⇒ **2+4+9+3 = 18** and … should ⇒ **7+2+4+9+3 = 25**.
  - We see **(+ (car L) X)** is a good … expression for ***this*** L!

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l)* ⇒ the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (let ((X (safe-sum (cdr L))))
        an expression that ⇒ value of (safe-sum L)
        and that involves X and, possibly, L          )))
```

- Suppose **L** ⇒ **(7 2 4 9 3)**, so **(cdr L)** ⇒ **(2 4 9 3)**.
  Then **X** ⇒ **2+4+9+3 = 18** and $\boxed{\ldots}$ should ⇒ **7+2+4+9+3 = 25**.
  - We see **(+ (car L) X)** is a good $\boxed{\ldots}$ expression for ***this*** **L**!

**Q.** For what non-null values of **L** is **(+ (car L) X)** a good $\boxed{\ldots}$ ?

**A.** **(+ (car L) X)** is a good $\boxed{\ldots}$ *when* **(car L)** *and* **X** ⇒ ***numbers***
(equivalently, *when* **(car L)** ⇒ ***a number*** *and* **X** ⇏ **ERR!**).

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22** ; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
 (defun safe-sum (L)
   (if (null L)
       0
       (let ((X (safe-sum (cdr L))))
         an expression that ⇒ value of (safe-sum L)
         and that involves X and, possibly, L        )))
```

**Q.** For what non-null values of **L** is **(+ (car L) X)** a good $\boxed{\;\dots\;}$ ?

**A.** **(+ (car L) X)** is a good $\boxed{\;\dots\;}$ *when* **(car L)** *and* **X** ⇒ ***numbers***
(equivalently, *when* **(car L)** ⇒ ***a number*** *and* **X** ⇏ **ERR!**).

**Q.** What is a good $\boxed{\;\dots\;}$ when **(car L)** ⇏ ***a number*** *or* **X** ⇒ **ERR!**?

**A.** A good $\boxed{\;\dots\;}$ expression in these cases is: **'ERR!**

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l)* ⇒ the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (let ((X (safe-sum (cdr L))))
         (if (and (numberp X) (numberp (car L)))
             (+ (car L) X)
             'ERR!))))
```

**Q.** For what non-null values of **L** is **(+ (car L) X)** a good $\boxed{\;\dots\;}$ ?

**A.** **(+ (car L) X)** is a good $\boxed{\;\dots\;}$ *when* **(car L)** *and* **X** ⇒ ***numbers***
(equivalently, *when* **(car L)** ⇒ ***a number*** *and* **X** ⇏ **ERR!**).

**Q.** What is a good $\boxed{\;\dots\;}$ when **(car L)** ⇏ ***a number*** *or* **X** ⇒ **ERR!**?

**A.** A good $\boxed{\;\dots\;}$ expression in these cases is: **'ERR!**

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l)* ⇒ the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (let ((X (safe-sum (cdr L))))
         (if (and (numberp X) (numberp (car L)))
             (+ (car L) X)
             'ERR! ))))
```

**Q.** Should we *eliminate the LET*?

**A.** No, because **X** is used ***twice*** in the case where

**(and (numberp X) (numberp (car L))) ⇒ T**

- ○ In this case **X** is used as the argument of **(numberp X)**, and used *again* as an argument of **(+ (car L) X)**!

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are **not** all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (let ((X (safe-sum (cdr L))))
         (if (and (numberp X) (numberp (car L)))
             (+ (car L) X)
             'ERR! ))))
```

**Q.** Is there a case that should be *moved outside the LET*?

**A.** **Yes:** The case **(numberp (car L)) ⇒ NIL** should be moved out.
There's ***no need to*** use **X** in that case, because the function should return ERR! regardless of the value of **X**.

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are* ***not*** *all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

So: **(safe-sum '(7 2 4 0 9)) ⇒ 22**; **(safe-sum '(7 2 A 9)) ⇒ ERR!**

```
(defun safe-sum (L)
  (if (null L)
      0
      (if (not (numberp (car L)))
        'ERR!
        (let ((X (safe-sum (cdr L))))
          (if ~~(and~~ (numberp X)~~(numberp (car L)))~~
              (+ (car L) X)
              'ERR! )))))
```

*copy*

**Q.** Is there a case that should be *moved outside the LET*?

**A.** **Yes:** The case **(numberp (car L)) ⇒ NIL** should be moved out.
There's ***no need to*** use **X** in that case.

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are* ***not*** *all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

```
(defun safe-sum (L)
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (let ((X (safe-sum (cdr L))))
            (if (numberp X)
                (+ (car L) X)
                'ERR!)))))
```

**1st version of the final definition.**

**Example** Write a function **safe-sum** such that:

- *If l ⇒ a proper list of numbers, then*
  *(safe-sum l) ⇒ the sum of the elements of that list.*
- *If l ⇒ a proper list whose elements are* ***not*** *all numbers, then*
  *(safe-sum l) ⇒* the symbol ERR!.

**2nd version of the final definition.**

```
(defun safe-sum (L)
  (cond ((null L) 0)
        ((not (numberp (car L))) 'ERR!)
        (t (let ((X (safe-sum (cdr L))))
             (cond ((numberp X) (+ (car L) X))
                   (t 'ERR!))))))
```

- We didn't eliminate the LET, as its local variable X is used *twice* in the case where each of (car L) and X ⇒ a number.
- Eliminating the LET would produce the function on the next slide, or an equivalent function that uses COND instead of nested IFs. Those functions would be ***extremely inefficient*** when L ⇒ a long list of numbers: Their running time grows ***exponentially*** with the length of the list.

- Eliminating LET from the 1st version of the definition gives:

```
(defun safe-sum (L); very slow if L ⇒ a long list of numbers!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          ~~(let ((X (safe-sum (cdr L))))~~
            (if (numberp ~~X~~ (safe-sum (cdr L)))
                (+ (car L) ~~X~~ (safe-sum (cdr L)))
                'ERR!)))))
```

- Eliminating LET from the 1st version of the definition gives:

```
(defun safe-sum (L); very slow if L ⇒ a long list of numbers!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

- Consider a call of **safe-sum** with argument value **(0 1 2 … 49)**.
- It makes **2**=**2**$^{1}$ recursive calls with argument value **(1 2 3 … 49)**.
- Each of those **2**$^{1}$ calls makes **2** recursive calls with argument value **(2 3 4 … 49)**, so there are a total of $2^1$×2=**2**$^{2}$ recursive calls with argument value **(2 3 4 … 49)**.
- Each of those **2**$^{2}$ calls makes **2** recursive calls with argument value **(3 4 5 … 49)**, so there are a total of $2^2$×2=**2**$^{3}$ recursive calls with argument value **(3 4 5 … 49)**.
- For $0 \le d \le 50$, there are **2**$^{d}$ calls with argument value **(*d* … 49)**.

- Eliminating LET from the 1st version of the definition gives:

```
 (defun safe-sum (L); very slow if L ⇒ a long list of numbers!
   (if (null L)
       0
       (if (not (numberp (car L)))
           'ERR!
           (if (numberp (safe-sum (cdr L)))
               (+ (car L) (safe-sum (cdr L)))
               'ERR!)))))
```

- Consider a call of **safe-sum** with argument value **(0 1 2 … 49)**.
- For $0 \le d \le 50$, there are $2^d$ calls with argument value **(*d* … 49)**.

  ∴ the *total* no. of *recursive* calls is $2^1 + \ldots + 2^{50} = 2^{51} - 2 > 2 \times 10^{15}$.
- **General Principle:** If a function f can make **2 or more direct recursive calls**, then a single call of f might well produce $2^d$ or more recursive calls of f at recursion depth *d*.
- **LET can be used to <u>*avoid*</u> making 2 or more direct recursive calls of a function *with the very same argument values*!**

- **General Principle:** If a function f can make **2 or more direct recursive calls**, then a single call of f might well produce $2^d$ or more recursive calls of f at recursion depth *d*.
- **LET can be used to *avoid* making 2 or more direct recursive calls of a function *with the very same argument values*!**
- The 1st and 2nd versions of **safe-sum** use **LET** in this way.

```
(defun safe-sum (L)
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (let ((X (safe-sum (cdr L))))
            (if (numberp X)
                (+ (car L) X)
                'ERR!))))))
```

**1st version of the final definition.**

- These versions never make more than one direct recursive call, as a result of which **(safe-sum '(0 1 … 49))** computes its result using just 50 recursive calls rather than quadrillions!

## Comments on Lisp Assignment 4

Problems 1–13 can be solved by starting with one of the templates below or a dual of the 2nd template in which the roles of e1 and e2 are switched. (These are just the templates presented earlier!)

```
(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 )))
```

```
(defun f (e1 e2)
  (if (null e1) or (zerop e1)
      value of (f nil e2) or (f 0 e2)
      (let ((X (f (cdr e1) e2) or (f (- e1 1) e2) ))
        an expression that ⇒ value of (f e1 e2) and
        that involves X and, possibly, e1 and/or e2 )))
```

## Comments on Lisp Assignment 4

Problems 1–13 can be solved by starting with one of the templates above or a dual of the 2nd template in which the roles of e1 and e2 are switched. (These are just the templates presented earlier!)

**Recall that:**

- 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***.

## An Interesting Advantage of Recursive Functions

- If *p* is a parameter of a recursive function **f** that has a smaller value in each recursive call than in the current call, then *a single call* of **f** that passes a large value to *p* will generally produce *many recursive calls* of **f** with smaller arguments.

- This can help to reveal bugs when testing **f**: A single test call of **f** with a large argument will typically also test **f** with many smaller arguments.

## Debugging Suggestions

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1 ⇏ NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

- You can use an analogous approach in other cases.
- We will assume the definition of f has the following form:

```
(defun f (e1 e2)
  (if (null e1)
      value of (f nil e2)
      (let ((X  (f (cdr e1) e2)))
       an expression that ⇒ value of (f e1 e2) and
       that involves X and, possibly, e1 and/or e2  )))
```

- However, a similar debugging approach can be used if the definition of f does not use LET (e.g., because the LET has been eliminated) or the definition has more than one base case before the LET.

## Debugging Suggestions

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** ⇏ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

1. Make sure you know what the base case **(f nil e2)** *should* return; test **f** to check that **(f nil e2)** always returns the right result: If it doesn't, fix the definition of **f** so it does!

2. Call **f** with different arguments. If for certain arguments *there's an evaluation error* or **f** *returns an incorrect result*, find arguments **e1** and **e2** such that:

   (i) **(f e1 e2)** ⇏ the correct result,

   ***but*** (ii) **(f (cdr e1) e2)** ⇒ the correct result.

   (ii) implies **(let ((X (f (cdr e1) e2)))** gives **X** the ***correct*** value, whereas (i) implies *the* $\boxed{\dots}$ expr ***doesn't*** *compute the correct result from* **X***'s value*!

   When you find arguments **e1** and **e2** that satisfy (i) & (ii), fix the $\boxed{\dots}$ expr so **(f e1 e2)** ⇒ the ***correct*** result.

**Repeat step 2** until you think the definition of **f** is correct.

## A Debugging Example Relating to Assignment 4

Problem 7 asks you to write a function PARTITION such that if *l* ⇒ a proper list of real numbers and *p* is a real number, then (PARTITION *l p*) returns a list whose CAR is a list of those elements of the list given by *l* that are ***less*** than *p*, and whose CADR is a list of the other elements of the list given by *l*. So:

**(partition () 4) ⇒ (NIL NIL)**   **(partition '(2 5 6 3) 5) ⇒ ((2 3) (5 6))**

Here is an ***incorrect*** definition that needs debugging:

```
(defun partition (L p)   ; Incorrect definition!
  (if (null L)
      '(()())
      (let ((X (partition (cdr L) p)))
        (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
              (t (list (cons (car L) (car X)) (cadr X)))))))
```

On *testing this function in Clisp*, we find:

- (partition () 4) ⇒ **(NIL NIL)**   Correct!
- (partition '(2 5 6 3) 5) ⇒ **((2 5 3) (6))** Wrong: should be
- (partition '(5 6 3) 5) ⇒ **((5 3) (6))** Wrong: should be **((3) (5 6))**
- (partition '(6 3) 5) ⇒ **((3) (6))**  Correct!

## A Debugging Example Relating to Assignment 4

Problem 7 asks you to write a function PARTITION such that if *l* ⇒ a proper list of real numbers and *p* is a real number, then (PARTITION *l p*) returns a list whose CAR is a list of those elements of the list given by *l* that are ***less*** than *p*, and whose CADR is a list of the other elements of the list given by *l*. So:

**(partition () 4) ⇒ (NIL NIL)** **(partition '(2 5 6 3) 5) ⇒ ((2 3) (5 6))**

Here is an ***incorrect*** definition that needs debugging:

```
(defun partition (L p)  ; Incorrect definition!
  (if (null L)
      '(()())
      (let ((X (partition (cdr L) p)))
        (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
              (t (list (cons (car L) (car X)) (cadr X)))))))
```

On *testing this function in Clisp*, we find:

- (partition '(5 6 3) 5) ⇒ **((5 3) (6))** Wrong: should be **((3) (5 6))**
- (partition '(6 3) 5) ⇒ **((3) (6))** Correct!

  When L ⇒ (5 6 3) and p ⇒ 5, we have that X ⇒ **((3) (6))**: *We must fix the* $\boxed{\dots}$ *expr* so it ⇒ **((3) (5 6))** [***instead of*** **((5 3) (6))**].