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)
      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))))
         (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 $\boxed{\ldots}$?

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

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

A. A good $\boxed{\ldots}$ 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 the 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
      (if (not (numberp (car L)))
          'ERR!
          (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.

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

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

```
(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!))))))
```

2nd version of the final definition.

- We didn't eliminate the LET, as its local variable X is used *twice* in the case where each of (car L) and X $\Rightarrow$ 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 is a 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 inefficient!
  (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 $2^1\times2=2^2$ 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 $2^2\times2=2^3$ 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 inefficient!
  (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 *prevent* a function from making 2 or more direct recursive calls *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 *prevent* a function from making 2 or more direct recursive calls *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***.

In the recursive function definitions that were given above:

- In non-base cases the result is computed using just one recursive call, and it is the same recursive call in all non-base cases.
- The function has a formal parameter e for which it passes the value of (cdr e) or (- e 1) to the same parameter of the recursive call in non-base cases.
  - e may not be the only parameter, but the value of any other parameter is passed ***without change*** to the same parameter of the recursive call in non-base cases.

All 13 problems in section 2 of **Lisp Assignment 4** can be solved using recursive functions of this simple kind, ***but when doing*** **Lisp Assignment 5** ***you must be prepared to write recursive functions that work differently***!

When a function makes a recursive call, there will often be a formal parameter e of the function for which the value passed to the same parameter of the recursive call is ***smaller in size*** than the value of e.

(cdr e) and (- e 1) may be used to produce the value of smaller size. ***Other*** expressions that can be used to do that include:

When a function makes a recursive call, there will often be a formal parameter e of the function for which the value passed to the same parameter of the recursive call is ***smaller in size*** than the value of e.

(cdr e) and (- e 1) may be used to produce the value of smaller size. ***Other*** expressions that can be used to do that include:

- (cddr e) if e ⇒ a nonempty list.
- (- e 2) if e ⇒ an integer ≥ 2.
- (floor e 2) if e ⇒ an integer other than 0 or -1.
  - (floor e 2) = ⌊e/2⌋ = e >> 1 in Java if e ⇒ an integer.

  (/ e 2) if e ⇒ an *even* integer other than 0.
- (cdr L1) if e ⇒ a nonempty list;
  here L1 ⇒ a list, obtained by ***transforming*** the list given by e in some way, whose length is ≤ the length of that list.
  - For Assignment 5, your function SSORT should use this kind of expression to produce the argument value for its recursive call.

Example of the Use of (cddr L) as a Recursive Call Argument

Recall from Assignment 4: If L ⇒ a list then (SPLIT-LIST L) returns a list of two lists, in which the 1st list consists of the 1st, 3rd, 5th, ... elements of the list given by L, and the 2nd list consists of the 2nd, 4th, 6th, ... elements of the list given by L.

For example: (SPLIT-LIST ( )) => (NIL NIL) (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

```
(defun split-list (L)
  (if (null L)
      '(()())
      (let ((X (split-list (cddr L))))
        an expression that ⇒ value of (split-list L)
        and that involves X and, possibly, L.           )))
```

- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and **...** should ⇒ **((A C 1 3 5) (B D 2 4))**.
- **Q.** What is a good **...** expression in this case?
  A. (list (cons (car L) (car X)) (cons (cadr L) (cadr X)))
- **Q.** For what non-null values of L is this ***not*** a good **...** ?

Example of the Use of (cddr L) as a Recursive Call Argument

```
(defun split-list (L)
  (if (null L)
      '(()())
      (let ((X (split-list (cddr L))))
        an expression that ⇒ value of (split-list L)
        and that involves X and, possibly, L.          )))
```

- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and $\boxed{\dots}$ should ⇒ **((A C 1 3 5) (B D 2 4))**.
- **Q.** What is a good $\boxed{\dots}$ expression in this case?
  A. (list (cons (car L) (car X)) (cons (cadr L) (cadr X)))
- **Q.** For what non-null values of L is this ***not*** a good $\boxed{\dots}$?
  A. It's ***not*** good if **L ⇒ a list of length 1**--e.g., **L ⇒ (B)**.
- **Q.** What is a good $\boxed{\dots}$ expression in ***that*** case?
  A. (list L ())

So $\boxed{\dots}$ can be written:

```
(cond ((null (cdr L)) (list L ()))
      (t (list (cons (car L) (car X))
               (cons (cadr L) (cadr X)))))
```

Example of the Use of (cddr L) as a Recursive Call Argument

```
(defun split-list (L)
  (if (null L)
      '(()())
      (let ((X (split-list (cddr L))))
        (cond ((null (cdr L)) (list L ()))
              (t (list (cons (car L) (car X))
                       (cons (cadr L) (cadr X)))))) )))
```

- As X is used twice in the **t** case, we must ***not*** eliminate the LET: The function would be very inefficient if it called **(split-list (cddr L))** twice!
- As X is ***not*** used in the **(null (cdr L))** case, it's good to move that case out of the LET.
- After that case is moved out of the LET, it can be combined with the **(null L)** base case, because **(list L ())** is a good value to return in both cases.

Example of the Use of (cddr L) as a Recursive Call Argument

```
(defun split-list (L)
  (if (null L)
      '(()())
      (let ((X (split-list (cddr L))))
        (cond ((null (cdr L)) (list L ()))
              (t (list (cons (car L) (car X))
                       (cons (cadr L) (cadr X))))))))
```

- As X is ***not*** used in the **(null (cdr L))** case, it's good to move that case out of the LET.
- After that case is moved out of the LET, it can be combined with the **(null L)** base case, because **(list L ())** is a good value to return in both cases.

Final version:

```
(defun split-list (L)
  (if (null (cdr L))
      (list L ())
      (let ((X (split-list (cddr L))))
        (list (cons (car L) (car X))
              (cons (cadr L) (cadr X))))))
```

Note that calling **(split-list (cddr L))** instead of **(split-list (cdr L))** ***reduces the depth of recursion***.