Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
-

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***<u>one</u>*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
- Suppose there are just 2 arguments and the ***<u>first</u>*** argument of the recursive call is the argument that has a different value from the corresponding argument of the current call.

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
- Suppose there are just 2 arguments and the ***first*** argument of the recursive call is the argument that has a different value from the corresponding argument of the current call. Then, assuming that argument ⇒ a proper list or nonnegative integer, we can often define the function as follows:

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
- Suppose there are just 2 arguments and the ***first*** argument of the recursive call is the argument that has a different value from the corresponding argument of the current call. Then, assuming that argument ⇒ a proper list or nonnegative integer, we can often define the function as follows:

```
(defun f (e1 e2)
  (if [                    ]
      [                           ]
      (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 ])))
```

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
- Suppose there are just 2 arguments and the ***first*** argument of the recursive call is the argument that has a different value from the corresponding argument of the current call. Then, assuming that argument ⇒ a proper list or nonnegative integer, we can often define the function as follows:

```
(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  )))
```

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
-

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
- Now suppose the ***second*** (rather than the first) argument of the recursive call is the argument that has a different value from the corresponding argument of the current call. Then, assuming that argument ⇒ a proper list or nonnegative integer, we can often define the function as follows:

Recursive Functions of More Than One Argument

- In simple definitions (such as the definitions you are expected to write for Lisp Assignment 4), only ***one*** of the arguments of the recursive call needs to have a different value from the corresponding argument of the current call.
- Now suppose the ***second*** (rather than the first) argument of the recursive call is the argument that has a different value from the corresponding argument of the current call. Then, assuming that argument ⇒ a proper list or nonnegative integer, we can often define the function as follows:

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

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

-
-
-
-

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- To solve this problem in the above-mentioned way, we must first decide whether it is the ***first*** or the ***second*** argument of the recursive call that will have a smaller value than the corresponding argument of 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- To solve this problem in the above-mentioned way, we must first decide whether it is the ***first*** or the ***second*** argument of the recursive call that will have a smaller value than the corresponding argument of the current call.
- Experienced programmers are able to "look ahead" and see which of these two possibilities leads to a good function definition, *but if you can't see which choice is right then just **guess***: If your guess doesn't yield a good definition, go back and make the other choice!
-
-

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- To solve this problem in the above-mentioned way, we must first decide whether it is the ***first*** or the ***second*** argument of the recursive call that will have a smaller value than the corresponding argument of the current call.
- Experienced programmers are able to "look ahead" and see which of these two possibilities leads to a good function definition, *but if you can't see which choice is right then just* ***guess***: If your guess doesn't yield a good definition, go back and make the other choice!
- We will attempt to write the function by giving the ***first*** argument of the recursive call a smaller value than the corresponding argument of 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- To solve this problem in the above-mentioned way, we must first decide whether it is the ***first*** or the ***second*** argument of the recursive call that will have a smaller value than the corresponding argument of the current call.
- Experienced programmers are able to "look ahead" and see which of these two possibilities leads to a good function definition, *but if you can't see which choice is right then just* ***guess***: If your guess doesn't yield a good definition, go back and make the other choice!
- We will attempt to write the function by giving the ***first*** argument of the recursive call a smaller value than the corresponding argument of the current call.
- This will turn out to be the right choice; we will see later why the other choice would ***not*** work.

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- We will attempt to write the function by giving the ***first*** argument of the recursive call a smaller value than the corresponding argument of 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- We will attempt to write the function by giving the ***first*** argument of the recursive call a smaller value than the corresponding argument of 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- We will attempt to write the function by giving the ***first*** argument of the recursive call a smaller value than the corresponding argument of the current call:

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

What will this be?

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

- We will attempt to write the function by giving the ***first*** argument of the recursive call a smaller value than the corresponding argument of the current call:

```
(defun append-2 (L1 L2)
  (if (null L1)                  value of (append-2 nil L2)
      L2
      (let ((X (append-2 (cdr L1) 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

```
(defun append-2 (L1 L2)
  (if (null L1)                value of (append-2 nil L2)
      L2
      (let ((X (append-2 (cdr L1) 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

```
(defun append-2 (L1 L2)
  (if (null L1)
      L2
      (let ((X (append-2 (cdr L1) 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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the ... expression, let's consider a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and *what* ...*'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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the ... expression, let's consider 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)**,

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)
So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the ... expression, let's consider 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)**,
  so **(cdr L1)** ⇒ and **X** ⇒ .
  For this L1 and L2, ... should ⇒ .

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the ... expression, let's consider 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ .
  For this L1 and L2, ... should ⇒ .

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the … expression, let's consider 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, … should ⇒ .

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the $\boxed{\ldots}$ expression, let's consider a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and what $\boxed{\ldots}$ *'s value should be in this case:*
- Suppose **L1** ⇒ **(1 2 3 4)** and **L2** ⇒ **(A B C)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q.

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

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

- To write the ... expression, let's consider 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, ... should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2) will ⇒ **(1 2 3 4 A B C)**? **Ans.:**

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)
So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

```
(defun append-2 (L1 L2)
  (if (null L1)
      L2
      (let ((X (append-2 (cdr L1) L2)))
```

an expression that ⇒ value of (append-2 L1 L2) and that involves X and, possibly, L1 and/or L2)))

- To write the $\boxed{\dots}$ expression, let's consider 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:*
- Suppose **L1** ⇒ **(1 2 3 4)** and **L2** ⇒ **(A B C)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\dots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2) will ⇒ **(1 2 3 4 A B C)**? **Ans.: (cons (car L1) X)**

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

```
(defun append-2 (L1 L2)
  (if (null L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

```
(defun append-2 (L1 L2)
  (if (null L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, ... should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

  Q.

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)

So: (append-2 '(1 2 3 4) '(A B C)) ⇒ (1 2 3 4 A B C)

```
(defun append-2 (L1 L2)
  (if (null L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, ... should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2) will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

  Q. Is **(cons (car L1) X)** a good ... expression for all valid values of L1 and L2 such that L1 ⇏ NIL?

  A. If we're not sure, try ***another*** pair of values of L1 & 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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, [...] should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, ... should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

- Suppose **L1** ⇒ **(A B C D E F)** and **L2** ⇒ **(1 2 3 4 5 6 7)**,
  so **(cdr L1)** ⇒ and **X** ⇒ .
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ .

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

- Suppose **L1** ⇒ **(A B C D E F)** and **L2** ⇒ **(1 2 3 4 5 6 7)**,
  so **(cdr L1)** ⇒ **(B C D E F)** and **X** ⇒ .
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ .

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, **...** should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.: (cons (car L1) X)**

- Suppose **L1** ⇒ **(A B C D E F)** and **L2** ⇒ **(1 2 3 4 5 6 7)**,
  so **(cdr L1)** ⇒ **(B C D E F)** and **X** ⇒ **(B C D E F 1 2 3 4 5 6 7)**.
  For this L1 and L2, **...** should ⇒ .

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, … should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.: (cons (car L1) X)**

- Suppose **L1** ⇒ **(A B C D E F)** and **L2** ⇒ **(1 2 3 4 5 6 7)**,
  so **(cdr L1)** ⇒ **(B C D E F)** and **X** ⇒ **(B C D E F 1 2 3 4 5 6 7)**.
  For this L1 and L2, … should ⇒ **(A B C D E F 1 2 3 4 5 6 7)**.
  ○

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, ... should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.: (cons (car L1) X)**

- Suppose **L1** ⇒ **(A B C D E F)** and **L2** ⇒ **(1 2 3 4 5 6 7)**,
  so **(cdr L1)** ⇒ **(B C D E F)** and **X** ⇒ **(B C D E F 1 2 3 4 5 6 7)**.
  For this L1 and L2, ... should ⇒ **(A B C D E F 1 2 3 4 5 6 7)**.
  - **(cons (car L1) X)** ⇒

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

- Suppose **L1** ⇒ **(A B C D E F)** and **L2** ⇒ **(1 2 3 4 5 6 7)**,
  so **(cdr L1)** ⇒ **(B C D E F)** and **X** ⇒ **(B C D E F 1 2 3 4 5 6 7)**.
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ **(A B C D E F 1 2 3 4 5 6 7)**.
  - **(cons (car L1) X)** ⇒ **(A B C D E F 1 2 3 4 5 6 7)** too. Good!

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\dots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

-

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 L1)
      L2
      (let ((X (append-2 (cdr L1) 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)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, … should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.: (cons (car L1) X)**

- When we are satisfied that **(cons (car L1) X)** is a good … expression for all valid values of L1 and L2 such that L1 ⇏ NIL, we complete the above definition!

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 L1)
      L2
      (let ((X (append-2 (cdr L1) L2)))
        (cons (car L1) X) )))
```

- Suppose **L1** ⇒ **(1 2 3 4)** and **L2** ⇒ **(A B C)**,
  so **(cdr L1)** ⇒ **(2 3 4)** and **X** ⇒ **(2 3 4 A B C)**.
  For this L1 and L2, $\boxed{\dots}$ should ⇒ **(1 2 3 4 A B C)**.

  Q. What expression (involving **X** and, possibly, L1 and/or L2)
  will ⇒ **(1 2 3 4 A B C)**? **Ans.:** **(cons (car L1) X)**

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 L1)
      L2
      (let ((X (append-2 (cdr L1) L2)))
        (cons (car L1) X))))
```

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 L1)
      L2
      (let ((X (append-2 (cdr L1) L2)))
        (cons (car L1) X))))
```

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

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

Final version:

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 L1)
      L2
      (let ((X (append-2 (cdr L1) L2)))
        (cons (car L1) X))))
```

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

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

Final version:

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

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

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

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.
-

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?

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?

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)
      (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 )))
```

What will this be?

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 ***try*** to write the ... expression, let's consider a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and what ... *'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)                    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 ***try*** to write the $\boxed{\ldots}$ expression, let's consider a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and what $\boxed{\ldots}$*'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 ***try*** to write the ... expression, let's consider a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and what ... *'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 ***try*** to write the $\boxed{\ldots}$ expression, let's consider a *possible pair of values of* L1 *and* L2, *the resulting value of* **X**, and *what* $\boxed{\ldots}$*'s value should be in this case:*
- Suppose **L1 ⇒ (1 2 3 4)** and **L2 ⇒ (A B C D E)**,
  so **(cdr L2)** ⇒ and **X** ⇒ .
  For this L1 and L2, $\boxed{\ldots}$ should ⇒ .
-

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 ***try*** to write the [...] expression, let's consider 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** ⇒ .
  For this L1 and L2, [...] should ⇒ .
-

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 ***try*** to write the ... expression, let's consider 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 ⇒ .
-

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 ***try*** to write the … expression, let's consider 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 <u>A</u> 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 ***try*** to write the ... expression, let's consider 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)**.
- There's ***no good way*** to construct **(1 2 3 4 A B C D E)** from **(1 2 3 4 B C D E)**, **(1 2 3 4)**, and **(A B C D E)**, so there's ***no good way*** to write ... !

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)**.
- There's ***no good way*** to construct **(1 2 3 4 A B C D E)** from **(1 2 3 4 B C D E)**, **(1 2 3 4)**, and **(A B C D E)**, so there's ***no good way*** to write ... !

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, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 <u>A</u> B C D E)**.
- There's ***no good way*** to construct **(1 2 3 4 <u>A</u> B C D E)** from **(1 2 3 4 B C D E)**, **(1 2 3 4)**, and **(A B C D E)**, so there's ***no good way*** to write $\boxed{\ldots}$!
-

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, $\boxed{\ldots}$ should ⇒ **(1 2 3 4 A B C D E)**.
- There's ***no good way*** to construct **(1 2 3 4 A B C D E)** from **(1 2 3 4 B C D E)**, **(1 2 3 4)**, and **(A B C D E)**, so there's ***no good way*** to write $\boxed{\ldots}$!
- So our original decision to let the ***second*** (rather than the first) argument of append-2 have the same value in the recursive call as in the current call was the right decision!

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**

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 <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))))
          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 $\boxed{\ldots}$ 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 ***eliminate the* LET**:

(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 <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**
**   ~~(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:

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***.

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 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 (numberp (car L)) (all-numbers (cdr 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! ***We've rewritten 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 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:**

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 **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!**

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**:

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 $\boxed{\dots}$ should ⇒ **7+2+4+9+3 = 25**.
  - We see **(+ (car L) X)** is a good $\boxed{\dots}$ 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**!

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*

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 $\boxed{\dots}$ should ⇒ **7+2+4+9+3 = 25**.
  - We see **(+ (car L) X)** is a good $\boxed{\dots}$ expression for ***this*** **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!**).

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 $\boxed{\ldots}$?

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 ... ?

A. **(+ (car L) X)** is a good ... *when* **(car L)** *and* **X** ⇒ ***numbers***
(equivalently, *when* **(car L)** ⇒ *a **number** and* **X** ⇏ **ERR!**).

Q.

A.

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 $\boxed{\ldots}$?

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.

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 $\boxed{\ldots}$?

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.

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.

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

○

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

Q.

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.

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:

- ○
- ○

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:

- **(numberp X)**'s value ***isn't needed***: **(and …) ⇒ NIL** regardless!
-

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:

- **(numberp X)**'s value ***isn't needed***: **(and …) ⇒ NIL** regardless!
- The value of **'ERR!** is returned, and **'ERR!** doesn't use **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.

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 ⇒ 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.