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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Importantly, n * (n-1)! = n! holds even when n = 1, as 0! = 1.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

OR

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

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

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

- The ... expression may have more than one case (as in problem **B** in Sec. 1 of Lisp Assignment 4): The ... expression may, e.g., be a **COND** or **IF** expression.
- If there is no case in which **X** is used more than once, then ***eliminate the LET***.
- If the LET isn't eliminated, ***move any case in which* X *needn't be used out of the*** LET. If the LET ***is*** eliminated but ***there's a case where the recursive call's result isn't needed, deal with such cases as base cases--i.e., without making a recursive call***.

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

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

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

- Note that the problem specification has this form:

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

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

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

  If *you drive on this road*, ***then*** *you must pay a toll.*

  This rule does ***<u>not</u>*** obligate you to do anything if you do ***<u>not</u>*** drive on the road in question: *It is logically impossible to violate this rule if you do not drive on the road!*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

-

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

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

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

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

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

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

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

  ○ For ***this*** **L**, what is a good […] expression? **Ans.: X**

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

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

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

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

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

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

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

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

```
(defun evens (L)
  (if (null L)

      nil

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

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

  Suppose **L** ⇒ **(4 2 -1 4 0 9 2 3)**, so **(cdr L)** ⇒ **(2 -1 4 0 9 2 3)**.
  Then **X** ⇒ **(2 4 0 2)** and [...] should ⇒ **(4 2 4 0 2)**.
  - For ***this*** **L**, what is a good [...] expression?
    Ans.: (cons (car L) **X**).
  - Is (cons (car L) **X**) a good [...] expression for ***all*** non-null values of **L** such that (***not*** (oddp (car L)))? **Ans. YES!**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.
- 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)) ⇒ (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!

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)        <- What will this be?
      (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)

- 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)
      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.:** **(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. 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)**

- 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)** ⇒ **(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, […] 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, […] 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))))
```

- **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))))
```