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**.

- Note that the problem specification has this form:

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

  This means our function will ***not*** be obligated to do anything in particular when its argument value is ***not*** 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 enter this exhibit, then you must buy a ticket.

  This rule does ***not*** obligate you to do anything if you do ***not*** enter the exhibit: *It is logically impossible to violate this rule if you do not enter the exhibit.*

- 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!

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.**

- 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!
- 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 ***<u>assume</u>*** to be satisfied).
- As a general rule, code that checks the validity of argument values should ***<u>not</u>*** 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 <u>omitting</u> 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 $\boxed{\dots}$ expression, let's first consider *<u>one</u> possible value of* **L**, *the resulting value of* **X**, and what $\boxed{\dots}$*'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 $\boxed{\dots}$ should ⇒ **(2 4 0 2)**.

  - For ***<u>this</u>*** **L**, what is a good $\boxed{\dots}$ expression? **Ans.: X**
  - Is **X** a good $\boxed{\dots}$ for ***<u>all</u>*** non-null values of **L**? If not, ***<u>when</u>*** is **X** a good $\boxed{\dots}$? **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 $\boxed{\ldots}$ if (oddp (car L)). To find a good $\boxed{\ldots}$ 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 $\boxed{\ldots}$ should ⇒ **(4 2 4 0 2)**.

  - For ***this*** **L**, what is a good $\boxed{\ldots}$ expression?
    Ans.: (cons (car L) **X**).
  - Is (cons (car L) **X**) a good $\boxed{\ldots}$ expression for ***all*** 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))))
        (cond ((oddp (car L)) X)
              (t (cons (car L) X)))))))
```

- We've seen that **X** is a good $\boxed{\dots}$ if (oddp (car L)).
- We've seen that (cons (car L) **X**) is a good $\boxed{\dots}$ if (***not*** (oddp (car L))).
- So now we can write $\boxed{\dots}$ as shown above!

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.**

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; 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)

```
(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)))
        (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))))
```

- 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)

```
(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 <u>A</u> B C D E)**.
- There's ***<u>no good way</u>*** 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 ***<u>no good way</u>*** to write ... !
- So our original decision to let the ***<u>second</u>*** (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 every element of the list is a number

(all-numbers l) ⇒ NIL otherwise.

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

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

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

      T

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

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

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

If l ⇒ a proper list, then

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

(all-numbers l) ⇒ NIL otherwise.

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

- **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 <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**

```
(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)))))
```