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

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

- 

- 

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

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

- We didn't eliminate the LET, as its local variable X is used *twice* in the case where each of (car L) and X ⇒ a number.
-

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

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

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

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

**(defun safe-sum (L) ; very inefficient!**<br>
**(if (null L)**<br>
**0**<br>
**(if (not (numberp (car L)))**<br>
**'ERR!**<br>
~~(let ((X (safe-sum (cdr L))))~~<br>
**(if (numberp ~~X~~ (safe-sum (cdr L)))**<br>
**(+ (car L) ~~X~~ (safe-sum (cdr L)))**<br>
**'ERR!)))))**

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!

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

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

-
-
-
-
-

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!))))
```

- Consider a call of **safe-sum** with argument value **(0 1 2 … 49)**.
-
-
-
-

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

- Consider a call of **safe-sum** with argument value **(0 1 2 … 49)**.
- It makes $2=2^1$ recursive calls with argument value **(1 2 3 … 49)**.
-
-
-

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

- Consider a call of **safe-sum** with argument value **(0 1 2 … 49)**.

- For $0 \le d \le 50$, there are $2^d$ calls with argument value **(*d* … 49)**.

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

  ∴ the *total* no. of *recursive* calls is $2^1 + \ldots + 2^{50} = 2^{51} - 2 > 2 \times 10^{15}$.
-
-

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

  ∴ the *total* no. of *recursive* calls is $2^1 + \ldots + 2^{50} = 2^{51} - 2 > 2 \times 10^{15}$.
- **General Principle:** If a function f can make **2 or more direct recursive calls**, then a single call of f might well produce $\mathbf{2^{\mathit{d}}}$ or more recursive calls of f at recursion depth *d*.
-

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

```
(defun safe-sum (L) ; very inefficient!
  (if (null L)
      0
      (if (not (numberp (car L)))
          'ERR!
          (if (numberp (safe-sum (cdr L)))
              (+ (car L) (safe-sum (cdr L)))
              'ERR!)))))
```

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

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

- **General Principle:** If a function f can make **2 or more direct recursive calls**, then a single call of f might well produce **$2^d$** or more recursive calls of f at recursion depth *d*.
- **LET can be used to *prevent* a function from making 2 or more direct recursive calls *with the very same argument values*!**

- **General Principle:** If a function f can make **2 or more direct recursive calls**, then a single call of f might well produce **$2^d$** or more recursive calls of f at recursion depth *d*.
- **LET can be used to *prevent* a function from making 2 or more direct recursive calls *with the very same argument values*!**
-
-

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

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

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

-

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

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

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

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

**Comments on Lisp Assignment 4**

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

**Comments on Lisp Assignment 4**

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

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

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

**Comments on Lisp Assignment 4**

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

**Recall that:**

- 

- 

**Comments on Lisp Assignment 4**

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

**Recall that:**

- If there is no case in which **X** is used more than once, then ***eliminate the LET***.
-

**Comments on Lisp Assignment 4**

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

**Recall that:**

- If there is no case in which **X** is used more than once, then ***eliminate the LET***.
- If the LET isn't eliminated, ***move any case in which X needn't be used out of the*** LET. If the LET ***is*** eliminated but ***there's a case where the recursive call's result isn't needed, deal with such cases as base cases--i.e., without making a recursive call***.

**Debugging Suggestions**

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

- 

- 

- 

## Debugging Suggestions

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

- You can use an analogous approach in other cases.
-
-

## Debugging Suggestions

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

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

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

## Debugging Suggestions

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

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

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

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

**Debugging Suggestions**

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

1.

2.

**Debugging Suggestions**

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

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

2.

**Debugging Suggestions**

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

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

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

   (i) **(f e1 e2)** $\not\Rightarrow$ the correct result,

   ***but*** (ii) **(f (cdr e1) e2)** $\Rightarrow$ the correct result.

## Debugging Suggestions

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

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

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

   (i) **(f e1 e2)** $\not\Rightarrow$ the correct result,

   ***but*** (ii) **(f (cdr e1) e2)** $\Rightarrow$ the correct result.

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

**Debugging Suggestions**

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

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

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

   (i) **(f e1 e2)** $\not\Rightarrow$ the correct result,

   ***but*** (ii) **(f (cdr e1) e2)** $\Rightarrow$ the correct result.

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

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

**Debugging Suggestions**

For concreteness, let's assume you are writing a 2-argument function **f** such that, when **e1** $\not\Rightarrow$ **NIL**, **(f e1 e2)** computes its result from **(f (cdr e1) e2)**.

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

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

   (i) **(f e1 e2)** $\not\Rightarrow$ the correct result,

   ***but*** (ii) **(f (cdr e1) e2)** $\Rightarrow$ the correct result.

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

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

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

## A Debugging Example Relating to Assignment 4

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

**A Debugging Example Relating to Assignment 4**

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

- (partition () 4) ⇒ **(NIL NIL)**    Correct!
-
-
-

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

**A Debugging Example Relating to Assignment 4**

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

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

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

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

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

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

**A Debugging Example Relating to Assignment 4**

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

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

## A Debugging Example Relating to Assignment 4

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

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

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

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

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

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

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

# A Debugging Example Relating to Assignment 4

```
(let ((X (partition (cdr L) p)))
  (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
        (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

## A Debugging Example Relating to Assignment 4

```
(let ((X (partition (cdr L) p)))
  (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
        (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

## A Debugging Example Relating to Assignment 4

```
    (let ((X (partition (cdr L) p)))
      (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
            (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

**Q.** When L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)),
***why*** does $\boxed{\dots}$ ⇒ the wrong result **((5 3) (6))**?

**A.**

- 

- 

# A Debugging Example Relating to Assignment 4

```
    (let ((X (partition (cdr L) p)))
     (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
           (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

**Q.** When L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)), ***why*** does $\boxed{\dots}$ ⇒ the wrong result **((5 3) (6))?**

**A.** Because the **(> (car L) p)** test of the 1st COND clause ⇒ **NIL**, so the result is given by the **t** clause, whose consequent form **(list (cons (car L) (car X)) (cadr X))** ⇒ **((5 3) (6))**.

-
-

# A Debugging Example Relating to Assignment 4

```
(let ((X (partition (cdr L) p)))
  (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
        (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

**Q.** When L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)),
***why*** does $\boxed{\ldots}$ ⇒ the wrong result **((5 3) (6))?**

**A.** Because the **(> (car L) p)** test of the 1st COND clause ⇒ **NIL**, so the result is given by the **t** clause, whose consequent form **(list (cons (car L) (car X)) (cadr X))** ⇒ **((5 3) (6))**.

- To get **((3) (5 6))** from L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6))
  **(list** ____________________ ____________________ **)** would work.
-

## A Debugging Example Relating to Assignment 4

```
(let ((X (partition (cdr L) p)))
  (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
        (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

**Q.** When L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)), ***why*** does $\boxed{\text{...}}$ ⇒ the wrong result **((5 3) (6))?**

**A.** Because the **(> (car L) p)** test of the 1st COND clause ⇒ **NIL**, so the result is given by the **t** clause, whose consequent form **(list (cons (car L) (car X)) (cadr X))** ⇒ **((5 3) (6))**.

- To get **((3) (5 6))** from L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)) **(list (car X) ______________ )** would work.
-

## A Debugging Example Relating to Assignment 4

```
    (let ((X (partition (cdr L) p)))
      (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
            (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

**Q.** When L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)),
***why*** does … ⇒ the wrong result **((5 3) (6))?**

**A.** Because the **(> (car L) p)** test of the 1st COND clause ⇒ **NIL**, so the result is given by the **t** clause, whose consequent form **(list (cons (car L) (car X)) (cadr X))** ⇒ **((5 3) (6))**.

- To get **((3) (5 6))** from L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6))
  **(list (car X) (cons (car L) (cadr X)))** would work.
-

# A Debugging Example Relating to Assignment 4

```
    (let ((X (partition (cdr L) p)))
      (cond ((> (car L) p) (list (car X) (cons (car L) (cadr X))))
            (t (list (cons (car L) (car X)) (cadr X))))))
```

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

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

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

**Q.** When L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)), ***why*** does $\boxed{\ldots}$ ⇒ the wrong result **((5 3) (6))?**

**A.** Because the **(> (car L) p)** test of the 1st COND clause ⇒ **NIL**, so the result is given by the **t** clause, whose consequent form **(list (cons (car L) (car X)) (cadr X))** ⇒ **((5 3) (6))**.

- To get **((3) (5 6))** from L ⇒ (5 6 3), p ⇒ 5, and X ⇒ ((3) (6)) **(list (car X) (cons (car L) (cadr X)))** would work.
- This is just ***the consequent form of the 1st COND clause***, so we can make $\boxed{\ldots}$ ⇒ **((3) (5 6))** by fixing that clause's test.

## Further Comments on Testing and Debugging Recursive Functions

**Further Comments on Testing and Debugging Recursive Functions**

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

-

## Further Comments on Testing and Debugging Recursive Functions

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

- This can be viewed as an advantage of recursion that makes it easier to discover bugs by testing: A single test call of **f** can generate very many other (recursive) calls of **f**.

# More Sophisticated Recursion

In the recursive function definitions that were given above:

-
-

In the recursive function definitions that were given above:

- In non-base cases the result is computed using just one recursive call, and it is the same recursive call in all non-base cases.
-

In the recursive function definitions that were given above:

- In non-base cases the result is computed using just one recursive call, and it is the same recursive call in all non-base cases.
- The function has a formal parameter e for which it passes the value of (cdr e) or (- e 1) to the same parameter of the recursive call in non-base cases.
  - 

In the recursive function definitions that were given above:

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

In the recursive function definitions that were given above:

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

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

In the recursive function definitions that were given above:

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

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

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

In the recursive function definitions that were given above:

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

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

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

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

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

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

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

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

- 
- 
- 
- 

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

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

- (cddr e) if e ⇒ a nonempty list.
-
-
-

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

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

- (cddr e) if e ⇒ a nonempty list.
- (- e 2) if e ⇒ an integer ≥ 2.
-
-

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

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

- (cddr e) if e ⇒ a nonempty list.
- (- e 2) if e ⇒ an integer ≥ 2.
- (floor e 2) if e ⇒ an integer other than 0 or -1.
  - (floor e 2) =

    =
-

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

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

- (cddr e) if e ⇒ a nonempty list.
- (- e 2) if e ⇒ an integer ≥ 2.
- (floor e 2) if e ⇒ an integer other than 0 or -1.
  - (floor e 2) = $\lfloor e/2 \rfloor$ = e >> 1 in Java if e ⇒ an integer
    = e/2 in Java if e ⇒ a non-negative integer.
-

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

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

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

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

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

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

  (/ e 2) if e ⇒ an *even* integer other than 0.
-

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

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

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

  (/ e 2) if e ⇒ an *even* integer other than 0.
- (cdr L1) if e ⇒ a nonempty list;
  here L1 ⇒ a list, obtained by ***transforming*** the list given by e in some way, whose length is ≤ the length of that list.
  -

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

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

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

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

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

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

For example: (SPLIT-LIST ( )) => (NIL NIL) (SPLIT-LIST '(B)) => ((B) NIL)

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

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST L) returns a list of two lists, in which the 1st list consists of the 1st, 3rd, 5th, ... elements of the list given by L, and the 2nd list consists of the 2nd, 4th, 6th, ... elements of the list given by L.
For example: (SPLIT-LIST ( )) => (NIL NIL) (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

•

•

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

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

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

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

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST L) returns a list of two lists, in which the $1^{st}$ list consists of the $1^{st}$, $3^{rd}$, $5^{th}$, ... elements of the list given by L, and the $2^{nd}$ list consists of the $2^{nd}$, $4^{th}$, $6^{th}$, ... elements of the list given by L.
For example: (SPLIT-LIST ( )) => (NIL NIL) (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

- 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**:
- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒
  and ... should ⇒ .

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST L) returns a list of two lists, in which the 1st list consists of the 1st, 3rd, 5th, ... elements of the list given by L, and the 2nd list consists of the 2nd, 4th, 6th, ... elements of the list given by L.
For example: (SPLIT-LIST ( )) => (NIL NIL) (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

- 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**:
- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and … should ⇒ .

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST  L) returns a list of two lists, in which the 1st list consists of the 1st, 3rd, 5th, ... elements of the list given by L, and the 2nd list consists of the 2nd, 4th, 6th, ... elements of the list given by L.
For example: (SPLIT-LIST ( )) => (NIL NIL)   (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

- 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**:
- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and ... should ⇒ **((A C 1 3 5) (B D 2 4))**.

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST  L) returns a list of two lists, in which the 1st list consists of the 1st, 3rd, 5th, ... elements of the list given by L, and the 2nd list consists of the 2nd, 4th, 6th, ... elements of the list given by L.
For example: (SPLIT-LIST ( )) => (NIL NIL)   (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and **...** should ⇒ **((A C 1 3 5) (B D 2 4))**.

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST  L) returns a list of two lists, in which the $1^{st}$ list consists of the $1^{st}$, $3^{rd}$,  $5^{th}$,  ...  elements of the list given by L, and the $2^{nd}$ list consists of the $2^{nd}$, $4^{th}$, $6^{th}$,  ... elements of the list given by L.
For example:  (SPLIT-LIST ( )) => (NIL NIL)    (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5))  => ((A C 1 3 5) (B D 2 4))

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

- Let L ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and **...** should ⇒ **((A C 1 3 5) (B D 2 4))**.
-

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

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

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

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

- Let **L** ⇒ **(A B C D 1 2 3 4 5)**, so **(cddr L)** ⇒ **(C D 1 2 3 4 5)**.
  Then **X** ⇒ **((C 1 3 5) (D 2 4))**
  and $\boxed{...}$ should ⇒ **((A C 1 3 5) (B D 2 4))**.
- **Q.** What is a good $\boxed{...}$ expression in this case?
  **A.**

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

**Recall from Assignment 4**: If L ⇒ a list then (SPLIT-LIST L) returns a list of two lists, in which the 1st list consists of the 1st, 3rd, 5th, ... elements of the list given by L, and the 2nd list consists of the 2nd, 4th, 6th, ... elements of the list given by L.
For example: (SPLIT-LIST ( )) => (NIL NIL) (SPLIT-LIST '(B)) => ((B) NIL)
(SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

- Let **L ⇒ (A B C D 1 2 3 4 5)**, so **(cddr L) ⇒ (C D 1 2 3 4 5)**.
  Then **X ⇒ ((C 1 3 5) (D 2 4))**
  and ... should ⇒ **((A C 1 3 5) (B D 2 4))**.
- **Q.** What is a good ... expression in this case?
  **A. (list (cons (car L) (car X)) (cons (cadr L) (cadr X)))**
- **Q.**

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

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

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

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

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

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

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

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

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

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

We want: (SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

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

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

We want: (SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

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

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

We want: (SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

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

  If L ⇒ (B), we ***want*** (split-list L) ⇒ ((B) NIL) but
  **(list … (cons (cadr L) (cadr X)))**
  ⇒ a list whose 2nd element is a CONS!

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

We want: (SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

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

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

We want: (SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

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

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

We want: (SPLIT-LIST '(A B C D 1 2 3 4 5)) => ((A C 1 3 5) (B D 2 4))

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

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

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

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

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

  **A. (list L ())**

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

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

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

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

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

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

So … can be written:

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

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

```
(defun split-list (L)
  (if (null L)
      '(()())
      (let ((X (split-list (cddr L))))

                                                         )))
```

So ... can be written:

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

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

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

- 
- 
- 

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

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

- As **X** is used twice in the **t** case, we must ***not*** eliminate the LET: The function would be very inefficient if it called **(split-list (cddr L))** twice!
-
-

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

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

- As X is used twice in the **t** case, we must ***not*** eliminate the LET: The function would be very inefficient if it called **(split-list (cddr L))** twice!
- As X is ***not*** used in the **(null (cdr L))** case, it's good to move that case out of the LET.
-

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

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

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

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

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

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

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

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

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

**Final version:**

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

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

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

**Final version:**

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

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

**Example of the Use of (floor n 2) as a Recursive Call Argument**

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately?

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

**Example of the Use of (floor n 2) as a Recursive Call Argument**

$e$ (i.e., the base of natural logs) is one of the best known constants. How can we calculate $e$ very accurately? To be concrete, let's say we want to find a number $y$ such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

One way is to use the following fact (which we'll assume is true but isn't very hard to prove if you know enough calculus):

$$\left(1+\frac{1}{n}\right)^{n} < e < \left(1+\frac{1}{n}\right)\left(1+\frac{1}{n}\right)^{n} = \left(1+\frac{1}{n}\right)^{n+1}$$

WolframAlpha computational intelligence.

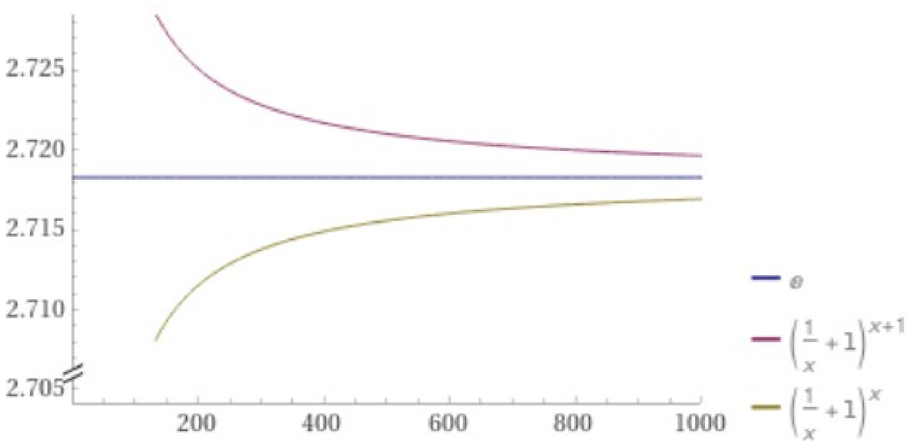
graph of e and (1+1/x)^(x+1) and (1+1/x)^x from 1 to 1000

Extended Keyboard   Upload   Examples

Input interpretation:

| | | |
|---|---|---|
| | $e$ | |
| plot | $\left(1+\frac{1}{x}\right)^{x+1}$ | $x = 1$ to $1000$ |
| | $\left(1+\frac{1}{x}\right)^{x}$ | |

Plot:



**Example of the Use of (floor n 2) as a Recursive Call Argument**

$e$ (i.e., the base of natural logs) is one of the best known constants. How can we calculate $e$ very accurately? To be concrete, let's say we want to find a number $y$ such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

One way is to use the following fact (which we'll assume is true but isn't very hard to prove if you know enough calculus):

$$\left(1+\frac{1}{n}\right)^{n} < e < \left(1+\frac{1}{n}\right)\left(1+\frac{1}{n}\right)^{n} = \left(1+\frac{1}{n}\right)^{n+1}$$

**Example of the Use of (floor n 2) as a Recursive Call Argument**

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

One way is to use the following fact (which we'll assume is true but isn't very hard to prove if you know enough calculus):

$$\left(1+\frac{1}{n}\right)^{n} < e < \left(1+\frac{1}{n}\right)\left(1+\frac{1}{n}\right)^{n} = \left(1+\frac{1}{n}\right)^{n+1}$$

When $n = 10^{25}$, this fact says that $(\clubsuit)$ holds when *y* is

$$(1+10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

## Example of the Use of (floor n 2) as a Recursive Call Argument

$e$ (i.e., the base of natural logs) is one of the best known constants. How can we calculate $e$ very accurately? To be concrete, let's say we want to find a number $y$ such that:

$$y < e < 1.000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

One way is to use the following fact (which we'll assume is true but isn't very hard to prove if you know enough calculus):

$$\left(1+\frac{1}{n}\right)^{n} < e < \left(1+\frac{1}{n}\right)\left(1+\frac{1}{n}\right)^{n} = \left(1+\frac{1}{n}\right)^{n+1}$$

When $n = 10^{25}$, this fact says that $(\clubsuit)$ holds when $y$ is

$$(1+10^{-25})^{10^{25}} = 1.000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) ⇒ $z^n$ if z ⇒ a number & n ⇒ an integer $\geq 0$

that can be used to compute $(1+10^{-25})^{10^{25}}$?

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

$(\clubsuit)$ holds when *y* is

$$(1+10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1+10^{-25})^{10^{25}}$?

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

It can be shown using calculus that ($\clubsuit$) holds when *y* is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow$ $z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1 + 10^{-25})^{10^{25}}$?

**Example of the Use of (floor n 2) as a Recursive Call Argument**

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

It can be shown using calculus that ($\clubsuit$) holds when *y* is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1 + 10^{-25})^{10^{25}}$?

- We ***cannot*** use a definition based on $z^n = z * z^{n-1}$ such as

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

It can be shown using calculus that (♣) holds when *y* is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) ⇒ $z^n$ if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1 + 10^{-25})^{10^{25}}$?

- We ***cannot*** use a definition based on $z^n = z * z^{n-1}$ such as

```
(defun power (z n)    ; far too inefficient!
  (cond ((= n 0) 1)
        (t (* z (power z (- n 1))))))
```

because

**Example of the Use of (floor n 2) as a Recursive Call Argument**

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

It can be shown using calculus that (♣) holds when *y* is

$$(1+10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) ⇒ $z^n$ if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1+10^{-25})^{10^{25}}$?

- We ***cannot*** use a definition based on $z^n = z * z^{n-1}$ such as

```
(defun power (z n)    ; far too inefficient!
  (cond ((= n 0) 1)
        (t (* z (power z (- n 1))))))
```

because when we pass $10^{25}$ to n this function would need a recursion depth of $10^{25}$, which would ***require an impossibly large amount of memory***; and it'd also ***take an impossibly long time*** to execute $10^{25}$ calls of power!

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

It can be shown using calculus that (♣) holds when *y* is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1 + 10^{-25})^{10^{25}}$?

•

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

It can be shown using calculus that (♣) holds when *y* is

$$(1+10^{-25})^{10^{25}} = 1.000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) ⇒ $z^n$ if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1+10^{-25})^{10^{25}}$?

- A solution is given by the function below, which is based on:

  $z^n = (z^{\lfloor n/2 \rfloor})^2$ if n is ***even***; $z^n = z*(z^{\lfloor n/2 \rfloor})^2$ if n is ***odd***.

  **Examples**: $z^{12} = (z^6)^2$ and $z^{11} = z*(z^5)^2$.

## Example of the Use of (floor n 2) as a Recursive Call Argument

*e* (i.e., the base of natural logs) is one of the best known constants. How can we calculate *e* very accurately? To be concrete, let's say we want to find a number *y* such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

It can be shown using calculus that (♣) holds when *y* is

$$(1+10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) ⇒ $z^n$ if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1+10^{-25})^{10^{25}}$?

- A solution is given by the function below, which is based on:

  $z^n = (z^{\lfloor n/2 \rfloor})^2$ if n is ***even***; $z^n = z*(z^{\lfloor n/2 \rfloor})^2$ if n is ***odd***.

  **Examples**: $z^{12} = (z^6)^2$ and $z^{11} = z*(z^5)^2$.

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

## Example of the Use of (floor n 2) as a Recursive Call Argument

we want to find a number $y$ such that:

$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$

It can be shown using calculus that $(\clubsuit)$ holds when $y$ is

$(1+10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$

**Q.** How can we write a recursive function **power** such that

(power z n) ⇒ $z^n$ if z ⇒ a number & n ⇒ an integer ≥ 0

that can be used to compute $(1+10^{-25})^{10^{25}}$?

- A solution is given by the function below:

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

## Example of the Use of (floor n 2) as a Recursive Call Argument

We want to find a number $y$ such that:

$$y < e < 1.0000000000000000000000001\,y = (1 + 10^{-25})\,y \qquad (\clubsuit)$$

It can be shown using calculus that $(\clubsuit)$ holds when $y$ is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow$ $z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1 + 10^{-25})^{10^{25}}$ ?

- A solution is given by the function below:

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

-
-

## Example of the Use of (floor n 2) as a Recursive Call Argument

We want to find a number $y$ such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

It can be shown using calculus that $(\clubsuit)$ holds when $y$ is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1 + 10^{-25})^{10^{25}}$ ?

- A solution is given by the function below:

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

- We get (floor n 2) by ***<u>chopping off the rightmost bit of</u>*** n.
- 

**Example of the Use of (floor n 2) as a Recursive Call Argument**

We want to find a number $y$ such that:

$$y < e < 1.0000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

It can be shown using calculus that ($\clubsuit$) holds when $y$ is

$$(1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$$

**Q.** How can we write a recursive function **power** such that

(power z n) $\Rightarrow$ $z^n$ if z $\Rightarrow$ a number & n $\Rightarrow$ an integer $\geq 0$

that can be used to compute $(1 + 10^{-25})^{10^{25}}$ ?

- A solution is given by the function below:

```
(defun power (z n)
  (cond ((zerop n) 1)
        (t (let ((X (power z (floor n 2))))
             (cond ((evenp n) (* X X))
                   (t (* z X X)))))))
```

- We get (floor n 2) by ***chopping off the rightmost bit of*** n.
- As $2^{83} < 10^{25} < 2^{84}$, the binary representation of $10^{25}$ has 84 bits: So a call of power with $10^{25}$ as the value of n makes a total of just 84 recursive calls!

This function **power** can now be used in Clisp to compute the number $y$ that satisfies $y < e < (1 + 10^{-25})\,y$.

```
euclid> cl
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.

[1]> (load "power.lsp")
;; Loading file power.lsp ...
;; Loaded file power.lsp
T
[2]> (setf (long-float-digits) 256)
256
[3]> (setf a (+ 1.0L0 (/ 1 (power 10 25))))
1.00000000000000000000000001L0
[4]> (power a (power 10 25))
2.71828182845904523536028733543857107480498532568559840479840654470561981531027L0
[5]>
```

Specifies that Clisp's LONG-FLOAT numbers are to have ⩾ 256 *binary* digits of precision.

1.0L0 means the long-float with value 1.0; this line sets a to the long-float with value $1 + 10^{-25}$.

This and earlier digits are the same as the corresponding digits of *e*.