More Sophisticated Recursion

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 usually be a parameter e of the function for which the value passed to the same parameter of the recursive call is ***smaller*** in size, for some suitable nonnegative integer measure of "size", than e's value.

(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 usually be a parameter e of the function for which the value passed to the same parameter of the recursive call is ***smaller*** in size, for some suitable nonnegative integer measure of "size", than e's value.

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

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

When a function makes a recursive call, there will usually be a parameter e of the function for which the value passed to the same parameter of the recursive call is ***smaller*** in size, for some suitable nonnegative integer measure of "size", than e's value.

(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))

```
(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))**.
- **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)                    (SPLIT-LIST '(B)) => ((B) NIL)
  (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 $\boxed{\text{...}}$ should ⇒ **((A C 1 3 5) (B D 2 4))**.
- **Q.** What is a good $\boxed{\text{...}}$ 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 $\boxed{\text{...}}$?
  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 2^{nd} 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))

(SPLIT-LIST '(B)) => ((B) NIL)

```
(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 ())

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))))) )))
```

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

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

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

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

Final version:

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

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

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:

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

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):

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

__Optional Exercise__ Students who have taken MATH 143 or MATH 152 may be interested in proving the above fact by showing the following:

- The left and right sides both converge to e as $n \to \infty$. (This follows from the Binomial Theorem and the power series for e^x.)
- The left side is a strictly increasing function of n. (This follows from the Binomial Theorem.)
- The right side is a strictly decreasing function of n, as the derivative of $(1+1/x)^{x+1}$ is negative for $x > 0$. (The latter follows from the fact that $\ln(1+1/x) < 1/x$ if $x > 0$ since $1+y < e^y$ if $y > 0$.)

WolframAlpha computational intelligence.

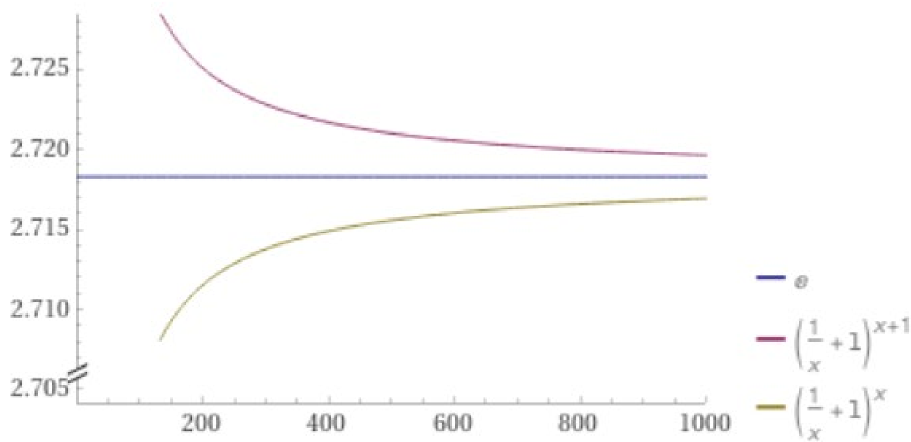
graph of e and (1+1/x)^(x+1) and (1+1/x)^x from 1 to 1000

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):

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

When $n = 10^{25}$, this fact says that ($\clubsuit$) holds when $y = (1+1/n)^n = (1 + 10^{-25})^{10^{25}} = 1.0000000000000000000000001^{10000000000000000000000000}$

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 ≥ 0

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

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:

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

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 $y = (1 + 10^{-25})^{10^{25}}$?

- 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:

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

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 $y = (1 + 10^{-25})^{10^{25}}$?

- 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:

$$y < e < 1.000000000000000000000000001\, y = (1 + 10^{-25})\, y \qquad (\clubsuit)$$

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

$$(1 + 10^{-25})^{10^{25}} = 1.000000000000000000000000001^{10000000000000000000000000}$$

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 ≥ 0

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

- 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 a 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 1L-25))
1.0000000000000000000000001L0
[4]> (power a (power 10 25))
2.718281828459045235360287335438571074804985325685598404798406544705619815310 27L0
[5]>
```

We want to compute $y = (1 + 10^{-25})^{10^{25}}$.

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

1L-25 means the long-float with value 10^{-25}; 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.

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

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)))))))
```

- In public-key cryptography one often needs to perform ***modular exponentiation***, whose goal is to compute m^n mod k for integers $m \geq 0$, $n \geq 0$, and $k \geq 1$; these integers may be very large. This can be done using a variant of the above function:

```
(defun power-mod (m n k) ; computes m^n mod k
  (cond ((zerop n) 1)
        (t (let ((X (power-mod m (floor n 2) k)))
             (cond ((evenp n) (mod (* X X) k))
                   (t (mod (* m X X) k)))))))
```

More Than One Formal Parameter of a Recursive Call May Have a Different Value from the Same Parameter of the Caller

- The **index** function in Assignment 5 illustrates this.
- Another illustration is provided by the exponentiation function below, which computes z^n using:

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

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

```
(defun pwr (z n)
  (cond ((zerop n) 1)
        ((evenp n) (pwr (* z z) (/ n 2)))
        (t (* z (pwr (* z z) (floor n 2))))))
```

- The following function performs ***modular*** exponentiation in an analogous way:

```
(defun pwr-mod (m n k) ; computes m^n mod k
  (cond
    ((zerop n) 1)
    ((evenp n) (pwr-mod (mod (* m m) k) (/ n 2) k))
    (t (mod (* m (pwr-mod (mod (* m m) k) (floor n 2) k)) k))))
```

Using Different Recursive Calls for Different Argument Values

Consider the MERGE-LISTS function of Assignment 5:

- MERGE-LISTS takes 2 arguments; ***each*** *argument value is assumed to be a proper* ***list of real numbers in ascending order***.
- (merge-lists L1 L2) returns a list that is equal to the list we would get if we sorted the list returned by (append L1 L2) into ascending order.

Two obvious recursive strategies to consider are:

1. Compute (merge-lists L1 L2) from (merge-lists (cdr L1) L2).
2. Compute (merge-lists L1 L2) from (merge-lists L1 (cdr L2)).

The function can indeed be written using these strategies, but *each of them is only good for* ***some*** *argument values*:

Example:

Using Different Recursive Calls for Different Argument Values

- MERGE-LISTS takes 2 arguments; ***each*** *argument value is assumed to be a proper* ***list of real numbers in ascending order***.
- (merge-lists L1 L2) returns a list that is equal to the list we would get if we sorted the list returned by (append L1 L2) into ascending order.

Two obvious recursive strategies to consider are:

1. Compute (merge-lists L1 L2) from (merge-lists (cdr L1) L2).
2. Compute (merge-lists L1 L2) from (merge-lists L1 (cdr L2)).

The function can indeed be written using these strategies, but *each of them is only good for* ***some*** *argument values*:

Example: L1 = (*2* **3 3 5 9 12**) L2 = (*8* **10 11 14**)

(merge-lists (cdr L1) L2) should ⇒ (**3 3 5** *8* **9 10 11 12 14**)
(merge-lists L1 (cdr L2)) should ⇒ (*2* **3 3 5 9 10 11 12 14**)
(merge-lists L1 L2) should ⇒ (*2* **3 3 5** *8* **9 10 11 12 14**)

Getting (merge-lists L1 L2) from (merge-lists (cdr L1) L2), L1, L2 is ***easy***!

Getting (merge-lists L1 L2) from (merge-lists L1 (cdr L2)), L1, L2 is ***hard***!

Strategy **1** is right ***in this example***, because **(car L1) < (car L2)**.

Using Different Recursive Calls for Different Argument Values

- MERGE-LISTS takes 2 arguments; ***each*** *argument value is assumed to be a proper* ***list of real numbers in ascending order***.
- (merge-lists L1 L2) returns a list that is equal to the list we would get if we sorted the list returned by (append L1 L2) into ascending order.

Two obvious recursive strategies to consider are:

1. Compute (merge-lists L1 L2) from (merge-lists (cdr L1) L2).
2. Compute (merge-lists L1 L2) from (merge-lists L1 (cdr L2)).

Example: L1 = (2 3 3 5 9 12) L2 = (8 10 11 14)

Strategy **1** is right ***in this example***, because **(car L1) < (car L2)**.

Example: L1 = (8 10 11 14) L2 = (2 3 3 5 9 12)

Strategy **2** is right ***in this example***, because **(car L2) < (car L1)**.

Example: L1 = (2 8 10 11 14) L2 = (2 3 3 5 9 12)

(merge-lists (cdr L1) L2) should ⇒ (2 3 3 5 8 9 10 11 12 14)
(merge-lists L1 (cdr L2)) should ⇒ (2 3 3 5 8 9 10 11 12 14)
(merge-lists L1 L2) should ⇒ (2 2 3 3 5 8 9 10 11 12 14)

(merge-lists (cdr L1) L2) and (merge-lists L1 (cdr L2)) are ***equal*** as **(car L2) = (car L1)**. Both strategies are good!