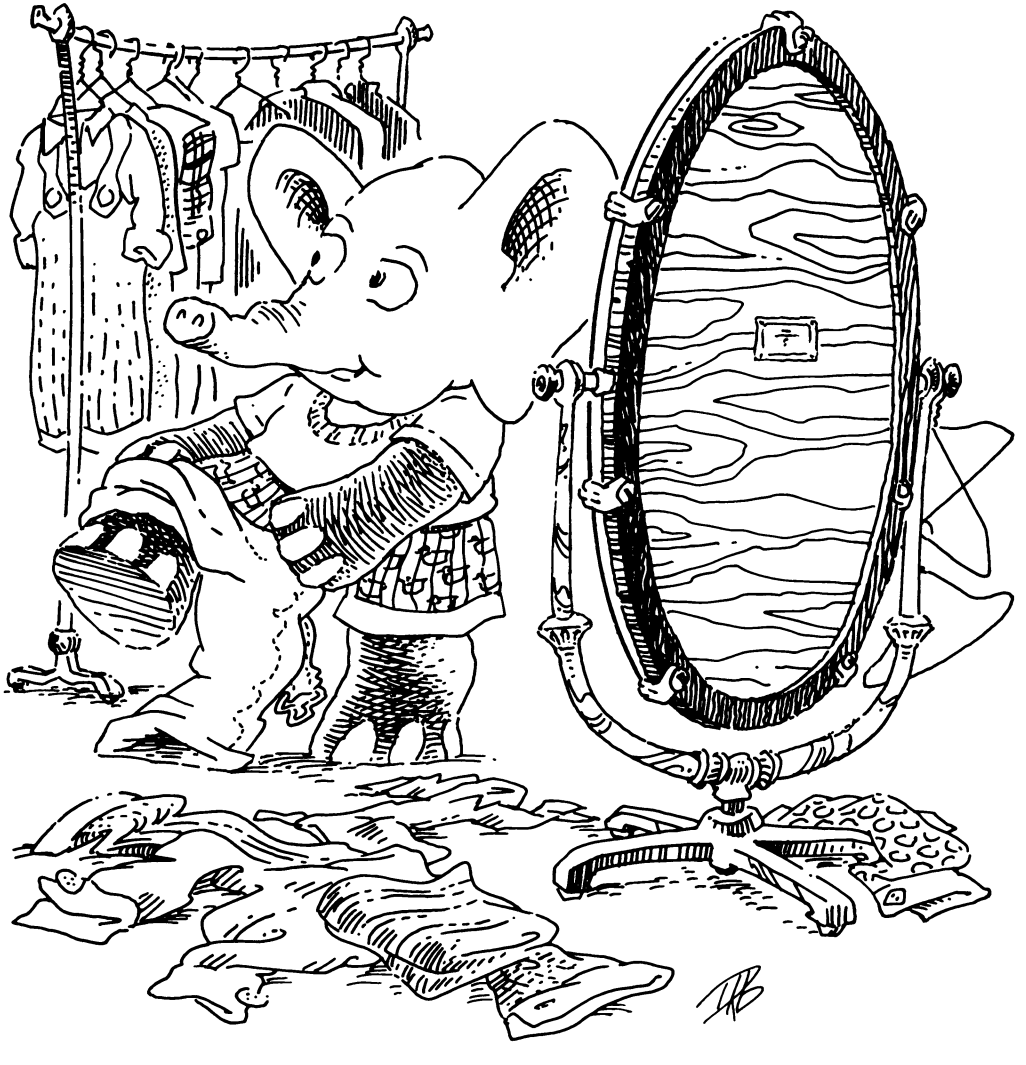
17.
We Change, Therefore We Are!



We didn't expect you so soon.

It is good to be back. What's next?

Let's look at *deep* again.

Here is a definition using (**if** ...):

```
(define deep
  (lambda (m)
    (if (zero? m)
        (quote pizza)
        (cons (deep (sub1 m))
              (quote ())))))
```

And let's look at *deepM* with the new version of *deep* included:

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (letrec
      ((D (lambda (m)
            (if (zero? m)
                (quote pizza)
                (cons (D (sub1 m))
                      (quote ())))))
       (lambda (n)
         (let ((exists (find n Ns RS)))
           (if (atom? exists)
               (let ((result (D n)))
                 (set! Rs (cons result Rs))
                 (set! Ns (cons n Ns))
                 result)
               exists))))))
```

Can you help D with its work?

Easy: D should refer to *deepM* instead of itself.

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (letrec
      ((D (lambda (m)
            (if (zero? m)
                (quote pizza)
                (cons (deepM (sub1 m))
                      (quote ())))))
       (lambda (n)
         (let ((exists (find n Ns RS)))
           (if (atom? exists)
               (let ((result (D n)))
                 (set! Rs (cons result Rs))
                 (set! Ns (cons n Ns))
                 result)
               exists))))))
```

Good. Is it true that there is no longer any need for (**letrec** ...) in *deepM*

Yes,
since D is no longer mentioned in the definition of D.

This means we can use (**let** ...)

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (let
      ((D (lambda (m)
            (if (zero? m)
                (quote pizza)
                (cons (deepM (sub1 m))
                  (quote ()))))))
      (lambda (n)
        (let ((exists (find n Ns RS)))
          (if (atom? exists)
              (let ((result (D n)))
                (set! Rs (cons result Rs))
                (set! Ns (cons n Ns))
                result)
              exists)))))))
```

Better: there needs to be only one (**let** ...)

Why?

Because *Ns* and *Rs* do not appear in the definition of *D*

This is true.

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ()))
        (D (lambda (m)
             (if (zero? m)
                 (quote pizza)
                 (cons (deepM (sub1 m))
                   (quote ()))))))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result (D n)))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

Can we replace the one use of D by the expression it names?

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result ...))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

What should we place at the dots?

Since the definition does not contain (**set!** D ...) and D is used in only one place, we can replace D by its value:

```
...
((lambda (m)
   (if (zero? m)
       (quote pizza)
       (cons (deepM (sub1 m))
             (quote ()))))
 n)
...
```

Therefore we can unname an expression that we named with the (**let** ...)

Yes, that is why the two definitions are equivalent.

Don't you think applying a (**lambda** ...) immediately to an argument is equivalent to (**let** ...)

Yes, determining the value of either one means determining the value of the value parts after associating a name with a value.

Complete the following definition of *deepM*

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result ...))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

```
...
(let ((m n))
  (if (zero? m)
      (quote pizza)
      (cons (deepM (sub1 m))
            (quote ()))))
...
```

Is it true that all we got was another (**let** ...)

And it introduced a name to name another name.

Is there a (**set!** *m* ...) in the value part of (**let** ((*m n*)) ...)

No. Are you asking whether we should unname again?

We could, couldn't we?

Yes, because now a name is replaced by a name.

Do it again!

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result ...))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

```
...
(if (zero? n)
    (quote pizza)
    (cons (deepM (sub1 n))
      (quote ())))
...
```

Wouldn't you like to know how much help *deepM* gives?

What does that mean?

Once upon a time, we wrote *deepM* to remember what values *deep* had for given numbers.

Oh, yes.

How many *cons*es does *deep* use to build pizza

None.

How many *cons*es does *deep* use to build (((((pizza)))))

Five, one for each topping.

How many *cons*es does *deep* use to build (((pizza)))

Three.

How many *cons*es does *deep* use to build pizza with a thousand toppings?

1000.

How many *cons*es does *deep* use to build all possible pizzas with at most a thousand toppings?

That's a big number:
the *cons*es of (*deep* 1000), and
the *cons*es of (*deep* 999), and
..., and
the *cons*es of (*deep* 0).

You mean 500,500?

Yes, thank you, Carl F. Gauss (1777–1855).

Yes, there is an easy way to determine this number, but we will show you the hard way. It is far more exciting.

Okay.

Guess what it is?

Can we write a function that determines it for us?

Yes, we can write the function *consC* which returns the same value as *cons* and counts how many times it sees arguments.

This is no different from writing *deepR* except that we use *add1* to build a number rather than *cons* to build a list.

```
(define consC
  (let ((N 0))
    (lambda (x y)
      (set! N (add1 N))
      (cons x y))))
```

Don't forget the imaginary name.

```
(define N1 0)
```

```
(define consC
  (lambda (x y)
    (set! N1 (add1 N1))
    (cons x y)))
```

Could we use this function to determine 500,500?

Sure, no problem.

How?

We just need to use *consC* instead of *cons* in the definition of *deep*:

```
(define deep
  (lambda (m)
    (if (zero? m)
        (quote pizza)
        (consC (deep (sub1 m))
               (quote ())))))
```

Wasn't this exciting?

Well, not really.

So let's see whether this new *deep* counts *cons*es

How about determining the value of (*deep* 5)?

That is easy; we shouldn't bother. What is the value of $\underline{N}_1$

We don't know, it is imaginary.

But that's how we count *cons*es

How could we possibly see something that is imaginary?

Here is one way.

```
(define counter)
```

```
(define consC
  (let ((N 0))
    (set! counter
      (lambda ()
        N))
    (lambda (x y)
      (set! N (add1 N))
      (cons x y))))
```

Is this as if we had written:

```
(define N_2 0)
```

```
(define counter
  (lambda ()
    N_2))
```

```
(define consC
  (lambda (x y)
    (set! N_2 (add1 N_2))
    (cons x y)))
```

Yes, what does *counter* refer to?

A function, perhaps?

Have we ever seen an incomplete definition before?

(**define** *counter*)

No, it looks strange.

It just means that we do not care what the first value of *counter* is, ...

... because we immediately change it?

Correct. But how many arguments does *counter* take?

None?

None!

So how do we use it?

What is the value of (*counter*)

It is whatever $\underline{N}_2$ refers to.

And what does $\underline{N}_2$ refer to?

At this time, 0.

What is the value of (*deep* 5)

(((((pizza))))).

What is the value of (*counter*)

5?

Yes, 5

How did that happen?
"Each time *consC* is used, one is added to $\underline{N}_2$. And the answer to (*counter*) always refers to whatever $\underline{N}_2$ refers to."

What is the value of (*deep* 7)

(((((((pizza)))))))).

What is the value of (*counter*)

Obvious: 12.

Is it clear now how we determine 500,500?

Not quite; we need to use *deep* on a thousand and one numbers.

But that is easy. Modify the function *supercounter* so that it returns the answer of (*counter*) when it has applied its argument to all the numbers between 0 and 1000

```
(define supercounter
  (lambda (f)
    (letrec
      ((S (lambda (n)
            (if (zero? n)
                (f n)
                (let ()
                  (f n)
                  (S (sub1 n)))))))
      (S 1000))))
```

As with (**let** ...) and (**lambda** ...), we can also have more than one expression in the value part of a (**letrec** ...):

```
(define supercounter
  (lambda (f)
    (letrec
      ((S (lambda (n)
            (if (zero? n)
                (f n)
                (let ()
                  (f n)
                  (S (sub1 n)))))))
      (S 1000)
      (counter))))
```

What is the value of (*supercounter f*) where *f* is *deep*

500512.

Is this what we expected?

No! We wanted 500500.

Where did the extra 12 come from?

Are these the leftovers from the previous experiments?

That's correct.

We should not have leftovers.

Let's get rid of them.

How?

Good question! Write a function *set-counter*

What does it do?

The function *set-counter* and *counter* are opposites. Instead of getting the value of the imaginary name, it sets it.

We could modify the definition of *consC*.

```
(define counter)
```

```
(define set-counter)
```

```
(define consC
  (let ((N 0))
    (set! counter
      (lambda ()
        N))
    (set! set-counter
      (lambda (x)
        (set! N x)))
    (lambda (x y)
      (set! N (add1 N))
      (cons x y))))
```

And what happens now?

We get three functions and an imaginary name:

```
(define N3 0)
```

```
(define counter
  (lambda ()
    N3))
```

```
(define set-counter
  (lambda (x)
    (set! N3 x)))
```

```
(define consC
  (lambda (x y)
    (set! N3 (add1 N3))
    (cons x y)))
```

Now, what is the value of (*set-counter* 0)

But?

It changed $\underline{N_3}$ to 0.

What is the value of (*supercounter f*) where *f* is *deep*

500500.

Is this what we expected?

Yes!

It is time to see how many *cons*es are used for (*deepM* 5)

Don't we need to modify its definition so that it uses *consC*?

Of course! What are you waiting for?

```
(define deepM
  (let ((Rs (quote ()))
        (Ns (quote ())))
    (lambda (n)
      (let ((exists (find n Ns RS)))
        (if (atom? exists)
            (let ((result
                   (if (zero? n)
                       (quote pizza)
                       (consC
                         (deepM (sub1 n))
                         (quote ())))))
              (set! Rs (cons result Rs))
              (set! Ns (cons n Ns))
              result)
            exists)))))
```

How many *cons*es does *deepM* use to build (((((pizza)))))

Probably five?

What is the value of (*counter*)

500505.

Yes!

Yes, but it means we forgot to initialize with *set-counter*.

What is the value of (*set-counter* 0)	
How many *cons*es does *deepM* use to build (((((pizza)))))	Five.
What is the value of (*counter*)	5.
What is the value of (*deep* 7)	(((((((pizza))))))).
What is the value of (*counter*)	Obvious: 7.
Didn't we need to *set-counter* to 0	No, we wanted to count the number of *cons*es that were needed to build (*deepM* 5) and (*deepM* 7).
Why isn't this 12	Because that was the point of *deepM*.
What is (*supercounter f*) where *f* is *deepM*	Don't we need to initialize?
No. What is (*supercounter f*) where *f* is *deepM*	1000.
How many more *cons*es does *deep* use to return the same value as *deepM*	499,500.
"A LISP programmer knows the value of everything but the cost of nothing."	Thank you, Alan J. Perlis (1922–1990).

But we know the value of food!

```
((((((((((((((((more pizza))))))))))))))))
 ((((((((((((((more pizza))))))))))))))
  ((((((((((((more pizza))))))))))))
   ((((((((((((more pizza))))))))))))
    ((((((((((more pizza))))))))))
     ((((((((((more pizza))))))))))
      ((((((((((more pizza))))))))))
       ((((((((more pizza))))))))
        (((((((more pizza)))))))
         (((((((more pizza)))))))
          ((((((more pizza))))))
           (((((more pizza)))))
            ((((more pizza))))
             (((more pizza)))
              ((more pizza))
               (more pizza)
                more pizza)
```

Here is *rember1** again:

```
(define rember1*
  (lambda (a l)
    (letrec
      ((R (lambda (l oh)
            (cond
              ((null? l)
               (oh (quote no)))
              ((atom? (car l))
               (if (eq? (car l) a)
                   (cdr l)
                   (cons (car l)
                     (R (cdr l) oh))))
              (else
                (let ((new-car
                        (letcc oh
                          (R (car l)
                            oh))))
                  (if (atom? new-car)
                      (cons (car l)
                        (R (cdr l) oh))
                      (cons new-car
                        (cdr l)))))))))
       (let ((new-l (letcc oh (R l oh))))
         (if (atom? new-l)
             l
             new-l))))))
```

Write it again using our counting version of *cons*

This is a safe version of the last definition we saw in chapter 14:

```
(define rember1*C
  (lambda (a l)
    (letrec
      ((R (lambda (l oh)
            (cond
              ((null? l)
               (oh (quote no)))
              ((atom? (car l))
               (if (eq? (car l) a)
                   (cdr l)
                   (consC (car l)
                     (R (cdr l) oh))))
              (else
                (let ((new-car
                        (letcc oh
                          (R (car l)
                            oh))))
                  (if (atom? new-car)
                      (consC (car l)
                        (R (cdr l) oh))
                      (consC new-car
                        (cdr l)))))))))
       (let ((new-l (letcc oh (R l oh))))
         (if (atom? new-l)
             l
             new-l))))))
```

What is the value of (*set-counter* 0)

What is the value of
  (*rember1*C a l*)
where
  a is noodles
and
  l is ((food) more (food))

((food) more (food)),
  because this list does not contain noodles.

And what is the value of (*counter*)

0,

because we never used *consC*. We always used the compass needle and the North Pole to get rid of pending *consC*es.

Do you also remember the first good version of *rember1**

```
(define rember1*
  (lambda (a l)
    (letrec
      ((R (lambda (l)
            (cond
              ((null? l) (quote ()))
              ((atom? (car l))
               (if (eq? (car l) a)
                   (cdr l)
                   (cons (car l)
                     (R (cdr l)))))
              (else
                (let ((av (R (car l))))
                  (if (eqlist? (car l) av)
                      (cons (car l)
                        (R (cdr l)))
                      (cons av
                        (cdr l)))))))))
       (R l))))
```

Rewrite it, too, using *consC*

It is the version that failed by repeatedly checking whether anything had changed for the *car* of a list that was a list:

```
(define rember1*C2
  (lambda (a l)
    (letrec
      ((R (lambda (l)
            (cond
              ((null? l) (quote ()))
              ((atom? (car l))
               (if (eq? (car l) a)
                   (cdr l)
                   (consC (car l)
                     (R (cdr l)))))
              (else
                (let ((av (R (car l))))
                  (if (eqlist? (car l) av)
                      (consC (car l)
                        (R (cdr l)))
                      (consC av
                        (cdr l)))))))))
       (R l))))
```

What is the value of (*set-counter* 0)

What is the value of

```
(consC (consC f (quote ()))
  (consC m
    (consC (consC f (quote ()))
      (quote ()))))
```

where

f is food

and

m is more

((food) more (food)).

What is the value of (counter)	5.
What is the value of (*set-counter* 0)	
(*rember1*C2 a l*) where *a* is noodles and *l* is ((food) more (food))	((food) more (food)), because this list does not contain noodles.
And what is the value of (*counter*)	5, because *rember1*C2* needs five *consC*s to rebuild the list ((food) more (food)).
What food are you in the mood for now?	Find a good restaurant that specializes in it and dine there tonight.