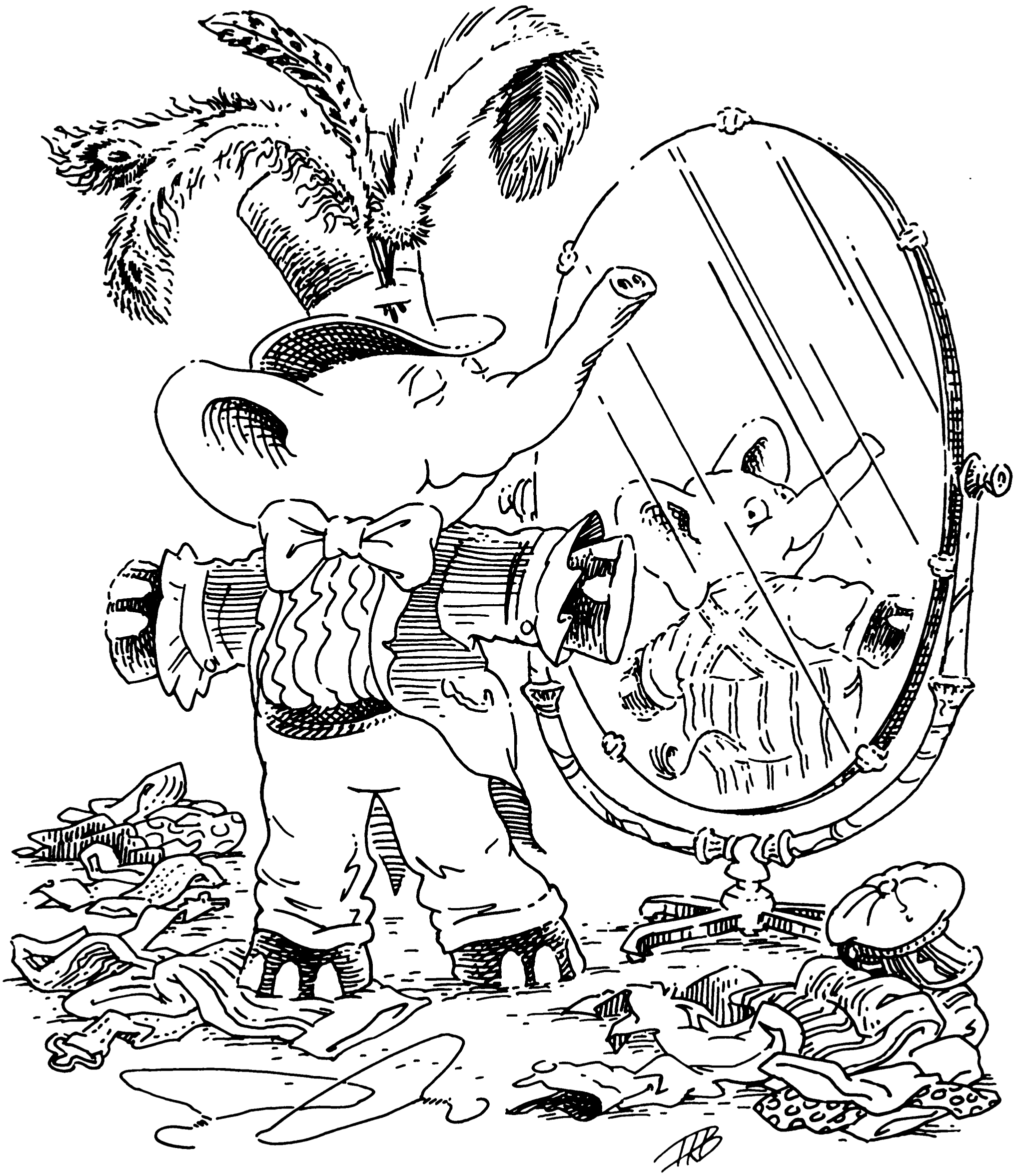
18.
We Change, Therefore We Are the Same !



What is the value of (*lots* 3)

(egg egg egg).

What is the value of (*lots* 5)

(egg egg egg egg egg).

What is the value of (*lots* 12)

(egg egg egg egg egg egg
egg egg egg egg egg egg).

What is the value of (*lenkth* (*lots* 3))

3.

What is the value of (*lenkth* (*lots* 5))

5.

What is the value of (*lenkth* (*lots* 15))

15.

Here is *lots*

```
(define lots
  (lambda (m)
    (cond
      ((zero? m) (quote ()))
      (else (kons[1] (quote egg)
                     (lots (sub1 m)))))))
```

[1] L, S: This is like **cons**.

And this is *lenkth*:

```
(define lenkth
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (lenkth (kdr[1] l)))))))
```

[1] L, S: This is like **cdr**.

How can we create a list of four eggs from (*lots* 3)

How about (*kons* (**quote** egg) (*lots* 3))?

Can we add an egg at the other end of the list?

Of course we can.

```
(define add-at-end
  (lambda (l)
    (cond
      ((null? (kdr l))
       (konsC (kar[1] l)
         (kons (quote egg)
           (quote ()))))
      (else (konsC (kar l)
              (add-at-end (kdr l)))))))
```

[1] L, S: This is like car.

Why do we ask (*null?* (*kdr l*))

Because we promise not to use *add-at-end* with non-empty lists.

What is a non-empty list?

A non-empty list is always created with *kons*. Its tail may be the empty list though.

What is *konsC*

konsC is to *consC* what *kons* is to *cons*.

What is the value of (*add-at-end* (*lots* 3))

(egg egg egg egg).

How many *konsC*es did we use?

The value of (*kounter*) is 3.

Can we add an egg at the end without making any new *kons*es except for the last one?

That would be a surprise!

Here is one way.

```
(define add-at-end-too
  (lambda (l)
    (letrec
      ((A (lambda (ls)
            (cond
              ((null? (kdr ls))
               (set-kdr[1] ls
                 (kons (quote egg)
                   (quote ()))))
              (else (A (kdr ls)))))))
      (A l)
      l)))
```

Are there any others?

Sure there are, but we are not interested in them.

Okay.

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

What is the value of (*kounter*)

0.

What is the value of
(*add-at-end-too* (*lots* 3))

(egg egg egg egg).

How many *konsC*es did *add-at-end-too* use?

Can we count them?

What if we told you that the value of (*kounter*) is 0

That's what it should be because *add-at-end-too* never uses *konsC* so the value of (*kounter*) should not change.

Do you remember *cons*

It is magnificent.

[1] L: This is like `rplacd`.
S: This is like `set-cdr!`.

Recall *zub1 edd1* and *sero?* from *The Little Schemer*. We can approximate *cons* in a similar way:

```
(define kons
  (lambda (kar kdr)
    (lambda (selector)
      (selector kar kdr))))
```

Write *kar* and *kdr*

```
(define kar
  (lambda (c)
    (c (lambda (a d) a))))
```

```
(define kdr
  (lambda (c)
    (c (lambda (a d) d))))
```

Suppose we had given you the definition of *bons*

```
(define bons
  (lambda (kar)
    (let ((kdr (quote ())))
      (lambda (selector)
        (selector
          (lambda (x) (set! kdr x))
          kar
          kdr)))))
```

Write *kar* and *kdr*

They are not too different from the previous definitions of *kar* and *kdr*.

```
(define kar
  (lambda (c)
    (c (lambda (s a d) a))))
```

```
(define kdr
  (lambda (c)
    (c (lambda (s a d) d))))
```

How can *bons* act like *kons*

Are we about to find out?

What is the value of (*bons e*) where *e* is egg

It is a function that is almost like (*kons e f*) where *f* is the empty list.

What is different?

When we determine the value of (*bons* (**quote egg**)), we also make a new imaginary name, $\underline{kdr}_1$. And the value that this imaginary name refers to can change over time.

How can we change the value that $\underline{kdr}_1$ refers to?

We could write a function that is almost like *kar* or *kdr*. This function could use the function (**lambda** (*x*) (**set!** $\underline{kdr}_1$ *x*)).

What is a good name for this function?

A good name is *set-kdr* and here is its definition.

```
(define set-kdr
  (lambda (c x)
    ((c (lambda (s a d) s)) x)))
```

Can we use *set-kdr* and *bons* to define *kons*

It's a little tricky but *bons* creates *kons*-like things whose *kdr* can be changed with *set-kdr*.

Let's do it!

Okay, this should do it:

```
(define kons
  (lambda (a d)
    (let ((c (bons a)))
      (set-kdr c d)
      c)))
```

Is *kons* a shadow of *cons*

It is.

Is *kons* different from *cons*

It certainly is. But don't forget that chapter 6 said: Beware of shadows.

Did we make any *kons*es when we added an egg to the end of the list?

Only for the new egg.

What is the value of

```
(define dozen (lots 12))
```

To find out, we must determine the value of (*lots* 12).

How many *kons*es did we use?

12.

What is the value of

```
(define bakers-dozen (add-at-end dozen))
```

To find out, we must determine the value of (*add-at-end dozen*).

How many *kons*es did we use now?

13.

How many *kons*es did we use altogether?

25.

What is the value of

```
(define bakers-dozen-too
  (add-at-end-too dozen))
```

To find out, we must determine the value of (*add-at-end-too dozen*).

How many *kons*es did we use now?

One.

How many *kons*es did we use altogether?

26.

Does that mean that the *kons*es in *dozen* are the same as the first twelve in *bakers-dozen-too*

Absolutely!

Does that mean that the *kons*es in *dozen* are the same as the first twelve in *bakers-dozen*

Absolutely not!

```
(define bakers-dozen-again
  (add-at-end dozen))
```

Okay.

How many *kons*es did we use now?

14.

Were you surprised that it wasn't 13?

Yes.

How many *kons*es did we use altogether?

40.

Does that mean that the *kons*es in *dozen* are the same as the first twelve in *bakers-dozen-again*

Absolutely not, again!

Does that mean that the *kons*es in *bakers-dozen* are the same as the first twelve in *bakers-dozen-again*

Absolutely not!

Does that mean that the *kons*es in *dozen* are still the same as the first twelve in *bakers-dozen-too*

It sure does!

What is the value of
  (*eklist? bakers-dozen bakers-dozen-too*)
where

```
(define eklist?
  (lambda (ls1 ls2)
    (cond
      ((null? ls1) (null? ls2))
      ((null? ls2) #f)
      (else
        (and (eq? (kar ls1) (kar ls2))
          (eklist? (kdr ls1) (kdr ls2)))))))
```

#t.

What does "the same" mean?

That is a deep philosophical question. Thank you, Gottfried W. Leibniz (1646–1716).

There is a new idea of "sameness" once we introduce (**set!** ...)

And that is?

Two *kons*es are the same if changing one changes the other.

What does that mean?

How can we change a *kons*

We defined *set-kdr* so that we could add a new egg at the end of the list ***without*** additional *kons*es.

Suppose we changed the first *kons* in *dozen*. Would it cause a change in the first *kons* of *bakers-dozen*

No.

Suppose again we changed the first *kons* in *dozen*. Would it cause a change in the first *kons* of *bakers-dozen-too*

Yes!

Time to define this notion of same.

```
(define same?
  (lambda (c1 c2)
    (let ((t1 (kdr c1))
          (t2 (kdr c2)))
      (set-kdr c1 1)
      (set-kdr c2 2)
      (let ((v (= (kdr c1) (kdr c2))))
        (set-kdr c1 t1)
        (set-kdr c2 t2)
        v))))
```

Thank you, Gerald J. Sussman and Guy L. Steele Jr.

What is the value of

```
  (same? bakers-dozen bakers-dozen-too)
```

#t.

Why?

The function *same?* temporarily changes the *kdr*s of two *kons*es. Then, if changing the second *kons* also affects the first *kons*, the two must be the same.

Could you explain this again?

If someone overate and you have a stomach ache, you are the one who ate too much.

How many imaginary names are used to determine the value of

```
  (same?
    (kons (quote egg) (quote ()))
    (kons (quote egg) (quote ())))
```

Two. One for the first *kons* and one for the second.

What is its value?

#f.

How did *same?* determine the answer?

The function first names the values of the *kdr*s. Then it changes them to different numbers. The answer is finally determined by comparing the values of the two *kdr*s. Finally, the *set-kdr*s change the respective *kdr*s so that they refer to their original values.

Here is the function *last-kons*

```
(define last-kons
  (lambda (ls)
    (cond
      ((null? (kdr ls)) ls)
      (else (last-kons (kdr ls))))))
```

Describe what it does.

The function *last-kons* returns the last *kons* in a non-empty *kons*-list.

```
(define long (lots 12))
```

Fine.

What does *long* refer to?

(egg egg egg egg egg egg
 egg egg egg egg egg egg).

What would be the value of
 (*set-kdr* (*last-kons long*) *long*)

Did you notice the subjunctive mood?

And then, what would be the value of
 (*lenkth long*)

No answer.

What is the value of
 (*set-kdr* (*last-kons long*) (*kdr* (*kdr long*)))

What is the value of
 (*lenkth long*)

Still no answer.

Why is there no value?

Because *long* is very long.

How many *kons*es does it contain?

12.

Didn't we write *length* together in *The Little Schemer*?

Yes, though *lenkth* now uses *kdr* because the lists it receives are made with *kons*.

Did we disobey any of the commandments when we wrote *length*

No, we didn't!

Then what's wrong?

The last *kons* of *long* no longer contains (**quote** ()) in the *kdr* part. Instead, the *kdr* part refers to some *kons* inside of *long*.

And?

No *kdr* refers to the empty list, because the only one that did was changed.

Why is this bad?

It means that *lenkth* keeps taking *kdr*s forever.

Draw a picture of "Kons the Magnificent" here.

Here is the function *finite-lenkth* which returns its argument's length, if it has one. If the argument doesn't have a length, the function returns false.

```
(define finite-lenkth
  (lambda (p)
    (letcc infinite
      (letrec
        ((C (lambda (p q)
              (cond
                ((same? p q)
                 (infinite #f))
                ((null? q) 0)
                ((null? (kdr q)) 1)
                (else
                  (+ (C (sl p) (qk q))
                     2)))))
         (qk (lambda (x) (kdr (kdr x))))
         (sl (lambda (x) (kdr x))))
        (cond
          ((null? p) 0)
          (else
            (add1 (C p (kdr p)))))))))
```

Bon appétit.

Guy's Favorite Pie

```
(define mongo
  (kons (quote pie)
    (kons (quote à)
      (kons (quote la)
        (kons (quote mode)
          (quote ()))))))
(set-kdr (kdr (kdr (kdr mongo))) (kdr mongo))
```