9 ... and Again, and Again, and Again, ...

DRAFT: September 18, 1995 – 09 : 28

Are you in the mood for caviar	Then we must go *looking* for it.
What is (*looking a lat*) where *a* is caviar and *lat* is (6 2 4 caviar 5 7 3)	#t, caviar is obviously in *lat*.
(*looking a lat*) where *a* is caviar and *lat* is (6 2 grits caviar 5 7 3)	#f.
Were you expecting something different?	Yes, caviar is still in *lat*.
True enough, but what is the first number in the lat?	6.
And what is the sixth element of *lat*	7.
And what is the seventh element?	3.
So *looking* clearly can't find caviar	True enough, because the third element is grits, which does not even resemble caviar.
Here is *looking* (**define** *looking* (**lambda** (*a lat*) (*keep-looking a* (*pick* 1 *lat*) *lat*))) Write *keep-looking*	We did not expect you to know this.
(*looking a lat*) where *a* is caviar and *lat* is (6 2 4 caviar 5 7 3)	#t, because (*keep-looking a* 6 *lat*) has the same answer as (*keep-looking a* (*pick* 1 *lat*) *lat*).

What is (*pick* 6 *lat*) where *lat* is (6 2 grits caviar 5 7 3)	7.
So what do we do?	(*keep-looking a* 7 *lat*) where *a* is caviar and *lat* is (6 2 4 caviar 5 7 3).
What is (*pick* 7 *lat*) where *lat* is (6 2 grits caviar 5 7 3)	3.
So what is (*keep-looking a* 3 *lat*) where *a* is caviar and *lat* is (6 2 4 caviar 5 7 3)	It is the same as (*keep-looking a* 4 *lat*).
Which is?	#t.
Write *keep-looking*	(**define** *keep-looking* (**lambda** (*a sorn lat*) (**cond** ((*number? sorn*) (*keep-looking a* (*pick sorn lat*) *lat*)) (**else** (*eq? sorn a*)))))
Can you guess what *sorn* stands for?	Symbol or number.
What is unusual about *keep-looking*	It does not recur on a part of *lat*.
We call this "unnatural" recursion.	It is truly unnatural.

Does *keep-looking* appear to get closer to its goal?	Yes, from all available evidence.
Does it always get closer to its goal?	Sometimes the list may contain neither caviar nor grits.
That is correct. A list may be a tup.	Yes, if we start *looking* in (7 2 4 7 5 6 3), we will never stop looking.
What is (*looking a lat*) where *a* is caviar and *lat* is (7 1 2 caviar 5 6 3)	This is strange!
Yes, it is strange. What happens?	We keep looking and looking and looking ...
Functions like *looking* are called partial functions. What do you think the functions we have seen so far are called?	They are called total.
Can you define a shorter function that does not reach its goal for some of its arguments?	(**define** *eternity* (**lambda** (*x*) (*eternity x*)))
For how many of its arguments does *eternity* reach its goal?	None, and this is the most unnatural recursion possible.
Is *eternity* partial?	It is the most partial function.
What is (*shift x*) where *x* is ((a b) c)	(a (b c)).

What is (*shift* *x*)
where
x is ((a b) (c d))

(a (b (c d))).

Define *shift*

This is trivial; it's not even recursive!

```
(define shift
  (lambda (pair)
    (build (first (first pair))
      (build (second (first pair))
        (second pair)))))
```

Describe what *shift* does.

Here are our words:
"The function *shift* takes a pair whose first component is a pair and builds a pair by shifting the second part of the first component into the second component."

Now look at this function:

```
(define align
  (lambda (pora)
    (cond
      ((atom? pora) pora)
      ((a-pair? (first pora))
       (align (shift pora)))
      (else (build (first pora)
                   (align (second pora)))))))
```

What does it have in common with *keep-looking*

Both functions change their arguments for their recursive uses but in neither case is the change guaranteed to get us closer to the goal.

Why are we not guaranteed that *align* makes progress?

In the second **cond**-line *shift* creates an argument for *align* that is not a part of the argument.

Which commandment does that violate?

The Seventh Commandment.

Is the new argument at least smaller than the original one?	It does not look that way.
Why not?	The function *shift* only rearranges the pair it gets.
And?	Both the result and the argument of *shift* have the same number of atoms.

Can you write a function that counts the number of atoms in *align*'s arguments?	No problem:
	(**define** *length**
	  (**lambda** (*pora*)
	  (**cond**
	  ((*atom? pora*) 1)
	  (**else**
	  (+ (*length** (*first pora*))
	  (*length** (*second pora*)))))))

Is *align* a partial function?	We don't know yet. There may be arguments for which it keeps aligning things.
Is there something else that changes about the arguments to *align* and its recursive uses?	Yes, there is. The first component of a pair becomes simpler, though the second component becomes more complicated.
In what way is the first component simpler?	It is only a part of the original pair's first component.
Doesn't this mean that *length** is the wrong function for determining the length of the argument? Can you find a better function?	A better function should pay more attention to the first component.
How much more attention should we pay to the first component?	At least twice as much.

Do you mean something like *weight**

```
(define weight*
  (lambda (pora)
    (cond
      ((atom? pora) 1)
      (else
        (+ (× (weight* (first pora)) 2)
           (weight* (second pora)))))))
```

That looks right.

What is (*weight** *x*)
where
x is ((a b) c)

7.

And what is (*weight** *x*)
where
x is (a (b c))

5.

Does this mean that the arguments get simpler?

Yes, the *weight**'s of *align*'s arguments become successively smaller.

Is *align* a partial function?

No, it yields a value for every argument.

Here is *shuffle* which is like *align* but uses *revpair* from chapter 7, instead of *shift*:

```
(define shuffle
  (lambda (pora)
    (cond
      ((atom? pora) pora)
      ((a-pair? (first pora))
       (shuffle (revpair pora)))
      (else (build (first pora)
                   (shuffle (second pora)))))))
```

The functions *shuffle* and *revpair* swap the components of pairs when the first component is a pair.

Does this mean that *shuffle* is total?

We don't know.

Let's try it. What is the value of (*shuffle* *x*) where
x is (a (b c))

(a (b c)).

(*shuffle* *x*) where
x is (a b)

(a b).

Okay, let's try something interesting. What is the value of (*shuffle* *x*) where
x is ((a b) (c d))

To determine this value, we need to find out what (*shuffle* (*revpair* *pora*)) is where
pora is ((a b) (c d)).

And how are we going to do that?

We are going to determine the value of
(*shuffle* *pora*)
where *pora* is ((c d) (a b)).

Doesn't this mean that we need to know the value of (*shuffle* (*revpair* *pora*)) where
(*revpair* *pora*) is ((a b) (c d))

Yes, we do.

And?

The function *shuffle* is not total because it now swaps the components of the pair again, which means that we start all over.

Is this function total?

```
(define C
  (lambda (n)
    (cond
      ((one? n) 1)
      (else
        (cond
          ((even? n) (C (÷ n 2)))
          (else (C (add1 (× 3 n)))))))))
```

It doesn't yield a value for 0, but otherwise nobody knows. Thank you, Lothar Collatz (1910–1990).

What is the value of $(A\ 1\ 0)$

2.

$(A\ 1\ 1)$

3.

$(A\ 2\ 2)$

7.

Here is the definition of A

```
(define A
  (lambda (n m)
    (cond
      ((zero? n) (add1 m))
      ((zero? m) (A (sub1 n) 1))
      (else (A (sub1 n)
               (A n (sub1 m)))))))
```

Thank you, Wilhelm Ackermann (1853–1946).

What does A have in common with *shuffle* and *looking*

A's arguments, like *shuffle*'s and *looking*'s, do not necessarily decrease for the recursion.

How about an example?

That's easy: $(A\ 1\ 2)$ needs the value of $(A\ 0\ (A\ 1\ 1))$. And that means we need the value of $(A\ 0\ 3)$.

Does A always give an answer?

Yes, it is total.

Then what is $(A\ 4\ 3)$

For all practical purposes, there is no answer.

What does that mean?

The page that you are reading now will have decayed long before we could possibly have calculated the value of $(A\ 4\ 3)$.

But answer came there none—
And this was scarcely odd, because
They'd eaten every one.

The Walrus and The Carpenter
—Lewis Carroll

Wouldn't it be great if we could write a function that tells us whether some function returns with a value for every argument?

It sure would. Now that we have seen functions that never return a value or return a value so late that it is too late, we should have some tool like this around.

Okay, let's write it.

It sounds complicated. A function can work for many different arguments.

Then let's make it simpler. For a warm-up exercise, let's focus on a function that checks whether some function stops for just the empty list, the simplest of all arguments.

That would simplify it a lot.

Here is the beginning of this function:

```
(define will-stop?
  (lambda (f)
    ...))
```

Can you fill in the dots?

What does it do?

Does *will-stop?* return a value for all arguments?

That's the easy part: we said that it either returns #t or #f, depending on whether the argument stops when applied to ().

Is *will-stop?* total then?

Yes, it is. It always returns #t or #f.

Then let's make up some examples. Here is the first one. What is the value of
(*will-stop? f*)
where
  f is *length*

We know that (*length l*) is 0
where *l* is ().

So?

Then the value of (*will-stop? length*) should be #t.

Absolutely. How about another example? What is the value of (*will-stop? eternity*)	(*eternity* (**quote** ())) doesn't return a value. We just saw that.
Does this mean the value of (*will-stop? eternity*) is #f	Yes, it does.
Do we need more examples?	Perhaps we should do one more example.

Okay, here is a function that could be an interesting argument for *will-stop?*
(**define** *last-try*
  (**lambda** (*x*)
  (**and** (*will-stop? last-try*)
  (*eternity x*))))
What is (*will-stop? last-try*)
	What does it do?

We need to test it on ()	If we want the value of (*last-try* (**quote** ())), we must determine the value of (**and** (*will-stop? last-try*) (*eternity* (**quote** ()))).
What is the value of (**and** (*will-stop? last-try*) (*eternity* (**quote** ())))	That depends on the value of (*will-stop? last-try*).
There are only two possibilities. Let's say (*will-stop? last-try*) is #f	Okay, then (**and** #f (*eternity* (**quote** ()))), is #f, since (**and** #f ...) is always #f.
So (*last-try* (**quote** ())) stopped, right?	Yes, it did.
But didn't *will-stop?* predict just the opposite?	Yes, it did. We said that the value of (*will-stop? last-try*) was #f, which really means that *last-try* will not stop.

So we must have been wrong about (*will-stop? last-try*)	That's correct. It must return #t , because *will-stop?* always gives an answer. We said it was total.
Fine. If (*will-stop? last-try*) is #t what is the value of (*last-try* (**quote** ()))	Now we just need to determine the value of (**and** #t (*eternity* (**quote** ()))), which is the same as the value of (*eternity* (**quote** ())).
What is the value of (*eternity* (**quote** ()))	It doesn't have a value. We know that it doesn't stop.
But that means we were wrong again!	True, since this time we said that (*will-stop? last-try*) was #t .
What do you think this means?	Here is our meaning: "We took a really close look at the two possible cases. If we can **define** *will-stop?*, then (*will-stop? last-try*) must yield either #t or #f. But it cannot—due to the very definition of what *will-stop?* is supposed to do. This must mean that *will-stop?* cannot be **define**d."
Is this unique?	Yes, it is. It makes *will-stop?* the first function that we can describe precisely but cannot **define** in our language.
Is there any way around this problem?	No. Thank you, Alan M. Turing (1912–1954) and Kurt Gödel (1906–1978).
What is (**define** ...)	This is an interesting question. We just saw that (**define** ...) doesn't work for *will-stop?*.

So what are recursive definitions?

Hold tight, take a deep breath, and plunge forward when you're ready.

Is this the function *length*

```
(define length
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))
```

It sure is.

What if we didn't have (**define** ...) anymore? Could we still define *length*

Without (**define** ...) nothing, and especially not the body of *length*, could refer to *length*.

What does this function do?

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (eternity (cdr l))))))
```

It determines the length of the empty list and nothing else.

What happens when we use it on a non-empty list?

No answer. If we give *eternity* an argument, it gives no answer.

What does it mean for this function that looks like *length*

It just won't give any answer for non-empty lists.

Suppose we could name this new function. What would be a good name?

$length_0$
because the function can only determine the length of the empty list.

How would you write a function that determines the length of lists that contain one or fewer items?

Well, we could try the following.

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (length_0 (cdr l))))))
```

Almost, but (**define** ...) doesn't work for $length_0$

So, replace $length_0$ by its definition.

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else
      (add1
        ((lambda (l)
           (cond
             ((null? l) 0)
             (else (add1
                     (eternity (cdr l))))))
         (cdr l))))))
```

And what's a good name for this function?

That's easy: $length_{\leq 1}$.

Is this the function that would determine the lenghts of lists that contain two or fewer items?

```
(lambda (l)
  (cond
    ((null? l) 0)
    (else
      (add1
        ((lambda (l)
           (cond
             ((null? l) 0)
             (else
               (add1
                 ((lambda (l)
                    (cond
                      ((null? l) 0)
                      (else
                        (add1
                          (eternity
                            (cdr l))))))
                  (cdr l))))))
         (cdr l))))))
```

Yes, this is $length_{\leq 2}$. We just replace *eternity* with the next version of *length*.

Now, what do you think recursion is?

What do you mean?

Well, we have seen how to determine the length of a list with no items, with no more than one item, with no more than two items, and so on. How could we get the function *length* back?

If we could write an infinite function in the style of $length_0$, $length_{\leq 1}$, $length_{\leq 2}$, ..., then we could write $length_\infty$, which would determine the length of all lists that we can make.

How long are the lists that we can make?

Well, a list is either empty, or it contains one element, or two elements, or three, or four, ..., or 1001, ...

But we can't write an infinite function.

No, we can't.

And we still have all these repetitions and patterns in these functions.

Yes, we do.

What do these patterns look like?

All these programs contain a function that looks like *length*. Perhaps we should abstract out this function: see The Ninth Commandment.

Let's do it!

We need a function that looks just like *length* but starts with (**lambda** (*length*) ...).

Do you mean this?

```
((lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l)))))))
 eternity)
```

Yes, that's okay. It creates $length_0$.

Rewrite $length_{\leq 1}$ in the same style.

```
((lambda (f)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (f (cdr l)))))))
 ((lambda (g)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (g (cdr l)))))))
  eternity))
```

Do we have to use *length* to name the argument?

No, we just used f and g. As long as we are consistent, everything's okay.

How about $length_{\leq 2}$

```
((lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l)))))))
 ((lambda (length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (length (cdr l)))))))
  ((lambda (length)
     (lambda (l)
       (cond
         ((null? l) 0)
         (else (add1 (length (cdr l)))))))
   eternity)))
```

Close, but there are still repetitions.

True. Let's get rid of them.

Where should we start?

Name the function that takes *length* as an argument and that returns a function that looks like *length*.

What's a good name for this function?

How about *mk-length* for "make *length*"?

Okay, do this to $length_0$

No problem.

```
((lambda (mk-length)
   (mk-length eternity))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

Is this $length_{\leq 1}$

```
((lambda (mk-length)
   (mk-length
    (mk-length eternity)))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

It sure is. And this is $length_{\leq 2}$.

```
((lambda (mk-length)
   (mk-length
    (mk-length
     (mk-length eternity))))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

Can you write $length_{\leq 3}$ in this style?

Sure. Here it is.

```
((lambda (mk-length)
   (mk-length
    (mk-length
     (mk-length
      (mk-length eternity)))))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

What is recursion like?

It is like an infinite tower of applications of *mk-length* to an arbitrary function.

Do we really need an infinite tower?

Not really of course. Everytime we use *length* we only need a finite number, but we never know how many.

Could we guess how many we need?

Sure, but we may not guess a large enough number.

When do we find out that we didn't guess a large enough number?

When we apply the function *eternity* that is passed to the innermost *mk-length*.

What if we could create another application of *mk-length* to *eternity* at this point?

That would only postpone the problem by one, and besides, how could we do that?

Well, since nobody cares what function we pass to *mk-length* we could pass it *mk-length* initially.

That's the right idea. And then we invoke *mk-length* on *eternity* and the result of this on the *cdr* so that we get one more piece of the tower.

Then is this still $length_0$

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               (length (cdr l))))))))
```

Yes, we could even use *mk-length* instead of *length*.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               (mk-length (cdr l))))))))
```

Why would we want to do that?

All names are equal, but some names are more equal than others.[1]

[1] With apologies to George Orwell (1903-1950).

True: as long as we use the names consistently, we are just fine.

And *mk-length* is a far more equal name than *length*. If we use a name like *mk-length*, it is a constant reminder that the first argument to *mk-length* is *mk-length*.

Now that *mk-length* is passed to *mk-length* can we use the argument to create an additional recursive use?

Yes, when we apply *mk-length* once, we get $length_{\leq 1}$

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               ((mk-length eternity)
                (cdr l))))))))
```

What is the value of

```
(((lambda (mk-length)
    (mk-length mk-length))
  (lambda (mk-length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1
                ((mk-length eternity)
                 (cdr l))))))))
 l)
```

where

l is (apples)

This is a good exercise. Work it out with paper and pencil.

Could we do this more than once?

Yes, just keep passing *mk-length* to itself, and we can do this as often as we need to!

What would you call this function?

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               ((mk-length mk-length)
                (cdr l))))))))
```

It is *length*, of course.

How does it work?

It keeps adding recursive uses by passing *mk-length* to itself, just as it is about to expire.

One problem is left: it no longer contains the function that looks like *length*

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               ((mk-length mk-length)
                (cdr l))))))))
```

Can you fix that?

We could extract this new application of *mk-length* to itself and call it *length*.

Why?

Because it really makes the function *length*.

How about this?

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length))))
```

Yes, this looks just fine.

Let's see whether it works.

Okay.

What is the value of

```
(((lambda (mk-length)
    (mk-length mk-length))
  (lambda (mk-length)
    ((lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else (add1 (length (cdr l)))))))
     (mk-length mk-length))))
 l)
```

where
l is (apples)

It should be 1.

First, we need the value of

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length))))
```

That's true, because the value of this expression is the function that we need to apply to l where
l is (apples)

So we really need the value of

```
((lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length)))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else (add1 (length (cdr l)))))))
    (mk-length mk-length))))
```

True enough.

But then we really need to know the value of

```
((lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l)))))))
 ((lambda (mk-length)
    ((lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else (add1 (length (cdr l)))))))
     (mk-length mk-length)))
  (lambda (mk-length)
    ((lambda (length)
       (lambda (l)
         (cond
           ((null? l) 0)
           (else (add1 (length (cdr l)))))))
     (mk-length mk-length)))))
```

Yes, that's true, too. Where is the end of this? Don't we also need to know the value of

```
((lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l)))))))
 ((lambda (length)
    (lambda (l)
      (cond
        ((null? l) 0)
        (else (add1 (length (cdr l)))))))
  ((lambda (mk-length)
     ((lambda (length)
        (lambda (l)
          (cond
            ((null? l) 0)
            (else (add1 (length (cdr l)))))))
      (mk-length mk-length)))
   (lambda (mk-length)
     ((lambda (length)
        (lambda (l)
          (cond
            ((null? l) 0)
            (else (add1 (length (cdr l)))))))
      (mk-length mk-length))))))
```

Yes, there is no end to it. Why?

Because we just keep applying *mk-length* to itself again and again and again ...

Is this strange?

It is because *mk-length* used to return a function when we applied it to an argument. Indeed, it didn't matter what we applied it to.

But now that we have extracted
(*mk-length mk-length*)
from the function that makes *length*
it does not return a function anymore.

No it doesn't. So what do we do?

Turn the application of *mk-length* to itself in our last correct version of *length* into a function:

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1
               ((mk-length mk-length)
                (cdr l)))))))))
```

How?

Here is a different way. If f is a function of one argument, is (**lambda** (x) (f x)) a function of one argument?

Yes, it is.

If (*mk-length mk-length*) returns a function of one argument, does

```
(lambda (x)
  ((mk-length mk-length) x))
```

return a function of one argument?

Actually,

```
(lambda (x)
  ((mk-length mk-length) x))
```

is a function!

Okay, let's do this to the application of *mk-length* to itself.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else
         (add1
           ((lambda (x)
              ((mk-length mk-length) x))
            (cdr l))))))))
```

Move out the new function so that we get *length* back.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   ((lambda (length)
      (lambda (l)
        (cond
          ((null? l) 0)
          (else
            (add1 (length (cdr l)))))))
    (lambda (x)
      ((mk-length mk-length) x)))))
```

Is it okay to move out the function?

Yes, we just always did the opposite by replacing a name with its value. Here we extract a value and give it a name.

Can we extract the function in the box that looks like *length* and give it a name?

Yes, it does not depend on *mk-length* at all!

Is this the right function?

```
((lambda (le)
   ((lambda (mk-length)
      (mk-length mk-length))
    (lambda (mk-length)
      (le (lambda (x)
            ((mk-length mk-length) x))))))
 (lambda (length)
   (lambda (l)
     (cond
       ((null? l) 0)
       (else (add1 (length (cdr l))))))))
```

Yes.

What did we actually get back?

We extracted the original function *mk-length*.

Let's separate the function that makes *length* from the function that looks like *length*

That's easy.

```
(lambda (le)
  ((lambda (mk-length)
     (mk-length mk-length))
   (lambda (mk-length)
     (le (lambda (x)
           ((mk-length mk-length) x))))))
```

Does this function have a name?

Yes, it is called the applicative-order Y combinator.

```
(define Y
  (lambda (le)
    ((lambda (f) (f f))
     (lambda (f)
       (le (lambda (x) ((f f) x)))))))
```

Does (**define** ...) work again?

Sure, now that we know what recursion is.

Do you now know why Y works?

Read this chapter just one more time and you will.

What is $(Y\ \ Y)$	Who knows, but it works very hard.
Does your hat still fit?	Perhaps not after such a mind stretcher.

Stop the World—I Want to Get Off.
Leslie Bricusse and Anthony Newley