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

Copyright © 1995 by The MIT Press.

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
We did not expect you to know this.

(define looking
  (lambda (a lat)
    (keep-looking a (pick 1 lat) lat)))

Write keep-looking

(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).

... and Again, and Again, and Again, ...
What is \( \text{pick} \ 6 \ \text{lat} \) where \\
\( \text{lat} \) is \( 6 2 \ \text{grits} \ \text{caviar} \ 5 7 3 \) \\

So what do we do? \( \text{(keep-looking a} \ 7 \ \text{lat)} \) where \( a \) is \( \text{caviar} \) and \\
\( \text{lat} \) is \( 6 2 4 \ \text{caviar} \ 5 7 3 \).

What is \( \text{pick} \ 7 \ \text{lat} \) where \\
\( \text{lat} \) is \( 6 2 \ \text{grits} \ \text{caviar} \ 5 7 3 \) \\

So what is \( \text{keep-looking a} \ 3 \ \text{lat} \) where \( a \) is \( \text{caviar} \) and \\
\( \text{lat} \) is \( 6 2 4 \ \text{caviar} \ 5 7 3 \) \\
It is the same as \( \text{(keep-looking a} \ 4 \ \text{lat)} \).

Which is? \#t.

Write \( \text{keep-looking} \) \\

\begin{verbatim}
(define keep-looking
 (lambda (a sorn lat)
   (cond
     ((number? sorn)
      (keep-looking a (pick sorn lat) lat))
     (else (eq? sorn a)))))
\end{verbatim}

Can you guess what \( \text{sorn} \) stands for? Symbol or number.

What is unusual about \( \text{keep-looking} \) It does not recur on a part of \( \text{lat} \).

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

... and Again, and Again, and Again, …
What is \( (\text{shift } x) \) where \( x = ((a\ b)\ (c\ d)) \)

Define \( \text{shift} \)

This is trivial; it's not even recursive!

\[
\begin{align*}
\text{(define shift} & \quad \text{(lambda (pair)} \\
& \quad \text{(build (first (first pair)} \\
& \quad \text{(build (second (first pair)} \\
& \quad \text{(second pair))))))
\end{align*}
\]

Describe what \( \text{shift} \) does.

Here are our words:

“The function \( \text{shift} \) 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:

\[
\begin{align*}
\text{(define align} & \quad \text{(lambda (pora)} \\
& \quad \text{(cond} \\
& \quad \quad ((\text{atom? pora})\ pora) \\
& \quad \quad ((\text{a-pair? (first pora)})) \\
& \quad \quad (\text{align (shift pora)})) \\
& \quad \quad (\text{else (build (first pora)} \\
& \quad \quad \text{(align (second pora)))))))))
\end{align*}
\]

What does it have in common with \textit{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 \( \text{align} \) makes progress?

In the second \textit{cond-line} \( \text{shift} \) creates an argument for \( \text{align} \) 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 \textit{shift} only rearranges the pair it gets.

And?  Both the result and the argument of \textit{shift} have the same number of atoms.

Can you write a function that counts the number of atoms in \textit{align}'s arguments?  No problem:

\begin{verbatim}
(define length*
  (lambda (pora)
    (cond
      ((atom?pora) 1)
      (else
        (+ (length* (firstpora))
          (length* (secondpora)))))))
\end{verbatim}

Is \textit{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 \textit{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 \textit{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.

... and \textit{Again, and Again, and Again,} ...
Do you mean something like \textit{weight*}?

\begin{verbatim}
(define weight*
  (lambda (pora)
    (cond
      ((atom? pora) 1)
      (else
       (+ (* (weight* (first pora)) 2)
          (weight* (second pora)))))))
\end{verbatim}

That looks right.

What is \textit{(weight* x)} where $x = ((a b) c)$?

\begin{itemize}
\item [5.] And what is \textit{(weight* x)} where $x = (a (b c))$?
\end{itemize}

Does this mean that the arguments get simpler?

Yes, the \textit{weight*}'s of \textit{align}'s arguments become successively smaller.

Is \textit{align} a partial function?

No, it yields a value for every argument.

Here is \textit{shuffle} which is like \textit{align} but uses \textit{revpair} from chapter 7, instead of \textit{shift}:

\begin{verbatim}
(define shuffle
  (lambda (pora)
    (cond
      ((atom? pora) pora)
      ((a-pair? (first pora))
       (shuffle (revpair pora)))
      (else (build (first pora)
                   (shuffle (second pora)))))))
\end{verbatim}

The functions \textit{shuffle} and \textit{revpair} swap the components of pairs when the first component is a pair.

Does this mean that \textit{shuffle} is total?

We don't know.
Let’s try it. What is the value of \((\text{shuffle } x)\) where \(x = (a\ b\ c)\).

\[(\text{shuffle } x)\]
where \(x = (a\ b)\)

Okay, let’s try something interesting. What is the value of \((\text{shuffle } x)\) where \(x = ((a\ b)\ (c\ d))\)?

To determine this value, we need to find out what \((\text{shuffle } \text{repair po})\) is where \(\text{po} = ((a\ b)\ (c\ d))\).

And how are we going to do that? We are going to determine the value of \((\text{shuffle po})\) where \(\text{po} = ((c\ d)\ (a\ b))\).

Doesn’t this mean that we need to know the value of \((\text{shuffle } \text{repair po})\) where \((\text{repair po}) = ((a\ b)\ (c\ d))\)?

Yes, we do.

And? The function \(\text{shuffle}\) is not total because it now swaps the components of the pair again, which means that we start all over.

Is this function total? It doesn’t yield a value for 0, but otherwise nobody knows. Thank you, Lothar Collatz (1910–1990).
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)))))))
```

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:

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

What does it do?

Can you fill in the dots?

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 \((\text{will-stop? eternity})\) \((\text{eternity (quote ())})\) doesn’t return a value. We just saw that.

Does this mean the value of \((\text{will-stop? eternity})\) 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 \text{will-stop?}

\[
\begin{align*}
(\text{define last-try} & \\
(\text{lambda} (x) & \\
(\text{and (will-stop? last-try)} & \\
(\text{eternity x)})))
\end{align*}
\]

What is \((\text{will-stop? last-try})\)

What does it do?

We need to test it on (). If we want the value of \((\text{last-try (quote ())})\), we must determine the value of

\[
(\text{and (will-stop? last-try)}
(\text{eternity (quote ())})
\]

What is the value of \((\text{and (will-stop? last-try)}\)

That depends on the value of \((\text{eternity (quote ())})\).

There are only two possibilities. Let’s say \((\text{will-stop? last-try})\) is \#f

Okay, then \((\text{and \#f (eternity (quote ())})\), is \#f, since \((\text{and \#f ...})\) is always \#f.

So \((\text{last-try (quote ())})\) stopped, right?

Yes, it did.

But didn’t \text{will-stop?} predict just the opposite?

Yes, it did. We said that the value of \((\text{will-stop? last-try})\) was \#f, which really means that \text{last-try} will not stop.
So we must have been wrong about $(\text{will-stop? last-try})$

That’s correct. It must return #t, because $\text{will-stop?}$ always gives an answer. We said it was total.

Fine. If $(\text{will-stop? last-try})$ is #t
what is the value of $(\text{last-try (quote ())})$

Now we just need to determine the value of
$(\text{and #t (eternity (quote ()))))$, which is the same as the value of $(\text{eternity (quote ())})$.

What is the value of $(\text{eternity (quote ())})$

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 $(\text{will-stop? last-try})$ 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 $\text{will-stop?}$, then
$(\text{will-stop? last-try})$ must yield either #t or #f. But it cannot—due to the very definition of what $\text{will-stop?}$ is supposed to do. This must mean that $\text{will-stop?}$ cannot be defined.”

Is this unique?

Yes, it is. It makes $\text{will-stop?}$ 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


What is $(\text{define ...})$

This is an interesting question. We just saw that $(\text{define ...})$ doesn’t work for $\text{will-stop?}$.
So what are recursive definitions? Hold tight, take a deep breath, and plunge forward when you’re ready.

Is this the function \textit{length}?

\begin{verbatim}
(define length
  (lambda (l)
    (cond
      ((null? l) 0)
      (else (add1 (length (cdr l)))))))
\end{verbatim}

It sure is.

What if we didn’t have \texttt{(define \ldots)} anymore? Could we still define \textit{length} without \texttt{(define \ldots)} nothing, and especially not the body of \textit{length}, could refer to \textit{length}.

What does this function do?

\begin{verbatim}
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (eternity (cdr l))))))
\end{verbatim}

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 \textit{eternity} an argument, it gives no answer.

What does it mean for this function that looks like \textit{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? \textit{length} 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.

\begin{verbatim}
(lambda (l)
  (cond
    ((null? l) 0)
    (else (add1 (length (cdr l))))))
\end{verbatim}
Almost, but (define ...) doesn’t work for \textit{length}_0

So, replace \textit{length}_0 by its definition.

\begin{verbatim}
(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))))))))

And what’s a good name for this function? That’s easy: \textit{length}_2.

Yes, this is \textit{length}_2. We just replace \textit{eternity} with the next version of \textit{length}.

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

\begin{verbatim}
(lambda (l)
  (cond
   ((null? l) 0)
   (else
    (add1
     ((lambda (l)
        (cond
          ((null? l) 0)
          (else
           (add1
            (eternity (cdr l)))))
             (cdr l))))))

Now, what do you think recursion is? What do you mean?

\ldots and Again, and Again, and Again, \ldots
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 \textit{length} back?

If we could write an infinite function in the style of \textit{length}_0, \textit{length}_1, \textit{length}_2, \ldots, then we could write \textit{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, \ldots, or 1001, \ldots

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 \textit{length}. Perhaps we should abstract out this function: see The Ninth Commandment.

Let’s do it!

We need a function that looks just like \textit{length} but starts with (\textbf{lambda} (\textit{length}) \ldots).

Do you mean this?

Yes, that’s okay. It creates \textit{length}_0.

```
((\textbf{lambda} (\textit{length})
  (\textbf{lambda} (\textit{l})
    (\textbf{cond}
      ((\textit{null?} \textit{l}) 0)
      (\textit{else} (\textit{add1 (length (\textit{cdr} \textit{l}))})))))

\textit{eternity})
```
Rewrite $\text{length}_{\leq 1}$ in the same style.

(((\text{lambda} (f))
  (\text{lambda} (l))
  (\text{cond}
    ((\text{null}?) 0)
    (\text{else} (\text{add}1 (f (\text{cdr} l))))))
((\text{lambda} (g))
  (\text{lambda} (l))
  (\text{cond}
    ((\text{null}?) 0)
    (\text{else} (\text{add}1 (g (\text{cdr} l))))))
\text{eternity})}

Do we have to use $\text{length}$ to name the argument? No, we just used $f$ and $g$. As long as we are consistent, everything’s okay.

How about $\text{length}_{\leq 2}$

(((\text{lambda} (\text{length}))
  (\text{lambda} (l))
  (\text{cond}
    ((\text{null}?) 0)
    (\text{else} (\text{add}1 (\text{length} (\text{cdr} l))))))
((\text{lambda} (\text{length}))
  (\text{lambda} (l))
  (\text{cond}
    ((\text{null}?) 0)
    (\text{else} (\text{add}1 (\text{length} (\text{cdr} l))))))
((\text{lambda} (\text{length}))
  (\text{lambda} (l))
  (\text{cond}
    ((\text{null}?) 0)
    (\text{else} (\text{add}1 (\text{length} (\text{cdr} l))))))
\text{eternity})

Close, but there are still repetitions. True. Let’s get rid of them.

Where should we start? Name the function that takes $\text{length}$ as an argument and that returns a function that looks like $\text{length}$.
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*₁? It sure is. And this is *length*₂.

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

Can you write *length*₃ in this style? Sure. Here it is.

```
((lambda (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 

\[
\begin{align*}
&((\text{lambda} \ (\text{mk-length})) \\
&\quad \quad \ (\text{mk-length} \ \text{mk-length})) \\
&((\text{lambda} \ (\text{length})) \\
&\quad \quad \ (\text{lambda} \ (l)) \\
&\quad \quad \quad \ (\text{cond} \\
&\quad \quad \quad \quad \ ((\text{null?} \ l) \ 0) \\
&\quad \quad \quad \quad \ (\text{else} \ (\text{add1} \\
&\quad \quad \quad \quad \quad \ (\text{length} \ (\text{cdr} \ l)))))))
\end{align*}
\]

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

\[
\begin{align*}
&((\text{lambda} \ (\text{mk-length})) \\
&\quad \quad \ (\text{mk-length} \ \text{mk-length})) \\
&((\text{lambda} \ (\text{mk-length})) \\
&\quad \quad \ (\text{lambda} \ (l)) \\
&\quad \quad \quad \ (\text{cond} \\
&\quad \quad \quad \quad \ ((\text{null?} \ l) \ 0) \\
&\quad \quad \quad \quad \ (\text{else} \ (\text{add1} \\
&\quad \quad \quad \quad \quad \ (\text{mk-length} \ (\text{cdr} \ l)))))))
\end{align*}
\]

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

Now that \textit{mk-length} is passed to \textit{mk-length} can we use the argument to create an additional recursive use?

Yes, when we apply \textit{mk-length} once, we get
\[
\text{\begin{verbatim}
((\lambda (\text{mk-length})
  (\text{mk-length} \text{mk-length}))
  (\lambda (\text{mk-length})
    (\lambda (l)
      (\text{cond}
        ((\text{null?} l) 0)
        (\text{else} (\text{addl}
          ((\text{mk-length} \text{eternity})
            (\text{cdr} l))))))))
\end{verbatim}
\]

What is the value of
\[
\text{\begin{verbatim}
((\lambda (\text{mk-length})
  (\text{mk-length} \text{mk-length}))
  (\lambda (\text{mk-length})
    (\lambda (l)
      (\text{cond}
        ((\text{null?} l) 0)
        (\text{else} (\text{addl}
          ((\text{mk-length} \text{eternity})
            (\text{cdr} l)))))))(l)
\end{verbatim}
\]

where
\[
l\text{ is (apples)}
\]

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

Could we do this more than once?

Yes, just keep passing \textit{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))))))))))
```

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

---

Can you fix that?  

---

Why?  

Because it really makes the function `length`.

... and Again, and Again, and Again, ...
How about this?

```scheme
((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

```scheme
(((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))))
```

It should be 1.

where

l is (apples)

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

\[
\begin{align*}
&\text{(lambda (mk-length)} \\
&\quad (\text{(lambda (length)} \\
&\quad \quad (\text{lambda (l)} \\
&\quad \quad \quad (\text{cond} \\
&\quad \quad \quad \quad ((\text{null? l}) 0) \\
&\quad \quad \quad \quad (\text{else (add1 (length (cdr l))))) \\
&\quad \quad (\text{mk-length mk-length)})) \\
&\quad (\text{(lambda (mk-length)} \\
&\quad (\text{(lambda (length)} \\
&\quad \quad (\text{lambda (l)} \\
&\quad \quad \quad (\text{cond} \\
&\quad \quad \quad \quad ((\text{null? l}) 0) \\
&\quad \quad \quad \quad (\text{else (add1 (length (cdr l))))) \\
&\quad \quad (\text{mk-length mk-length)})) \\
&\quad \text{True enough.}
\end{align*}
\]

But then we really need to know the value of

\[
\begin{align*}
&\text{(lambda (length)} \\
&\quad (\text{lambda (l)} \\
&\quad \quad (\text{cond} \\
&\quad \quad \quad ((\text{null? l}) 0) \\
&\quad \quad \quad (\text{else (add1 (length (cdr l))))) \\
&\quad \text{Yes, that’s true, too. Where is the end of this? Don’t we also need to know the value of}
\end{align*}
\]

\[
\begin{align*}
&\text{(lambda (length)} \\
&\quad (\text{lambda (l)} \\
&\quad \quad (\text{cond} \\
&\quad \quad \quad ((\text{null? l}) 0) \\
&\quad \quad \quad (\text{else (add1 (length (cdr l))))) \\
&\quad \text{of}
\end{align*}
\]

\[
\begin{align*}
&\text{(lambda (mk-length)} \\
&\quad (\text{(lambda (length)} \\
&\quad \quad (\text{lambda (l)} \\
&\quad \quad \quad (\text{cond} \\
&\quad \quad \quad \quad ((\text{null? l}) 0) \\
&\quad \quad \quad \quad (\text{else (add1 (length (cdr l))))) \\
&\quad \quad (\text{mk-length mk-length)})) \\
&\quad \text{... and Again, and Again, and Again, ...}
\end{align*}
\]
Yes, there is no end to it. Why?

Because we just keep applying \textit{mk-length} to itself again and again and again \ldots

Is this strange?

It is because \textit{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

\[
(\text{mk-length \ mk-length})
\]

from the function that makes \textit{length} it does not return a function anymore.

No it doesn’t. So what do we do?

Turn the application of \textit{mk-length} to itself in our last correct version of \textit{length} into a function:

\[
((\lambda (\text{mk-length})
  (\text{mk-length \ mk-length}))

(\lambda (\text{mk-length})
  (\lambda (l)
    (\text{cond}
      ((\text{null?} \ l) 0)
      (\text{else} \ (\text{add1}
        ((\text{mk-length \ mk-length})
          (\text{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 \((\text{mk-length \ mk-length})\) returns a function of one argument, does

\[
(\lambda (x)
  ((\text{mk-length \ mk-length}) \ x))
\]

return a function of one argument?

Actually,

\[
(\lambda (x)
  ((\text{mk-length \ mk-length}) \ x))
\]

is a function!
Okay, let's do this to the application of \( mk\text{-}length \) to itself.

\[
\begin{align*}
\text{((- Frauen abs) (mk-length))} \\
\text{((mk-length mk-length))} \\
\text{(mk-length mk-length)} \\
\text{(mk-length l)} \\
\text{(cond)} \\
\text{((null? l) 0)} \\
\text{(else)} \\
\text{(add1)} \\
\text{((lambda (x)))} \\
\text{((mk-length mk-length) x))} \\
\text{(cdr l)))))))))))
\end{align*}
\]

Move out the new function so that we get \( length \) back.

\[
\begin{align*}
\text{((- Frauen abs) (mk-length))} \\
\text{((mk-length mk-length))} \\
\text{(mk-length mk-length)} \\
\text{(mk-length l)} \\
\text{(cond)} \\
\text{((null? l) 0)} \\
\text{(else)} \\
\text{(add1)} \\
\text{(lambda (x))} \\
\text{((mk-length mk-length) x))} \\
\end{align*}
\]

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 \( mk\text{-}length \) at all!
Is this the right function?  

Yes.

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

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.

```
(l lama (le)
   ((l la m a (mk-length)
      (mk-length mk-length)))
(l lambda (mk-length)
   (l e (l ama (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

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