A Lecture on the *Why* of *Y*

by Matthias Felleisen

Is this the function *length*?

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

It sure is.

Suppose (**define** ... ...) no longer works. Can you describe in your own words what *length* does?

For one, the body of *length* cannot refer to *length*.

Then we might as well write something like this.

```
(define length
  (lambda (l)
    (if (null? l) 0
        (add1 (hukairs (cdr l))))))
```

Yes, except that (**define** ... ...) doesn't work anymore.

So perhaps something more like this?

```
(lambda (l)
  (if (null? l) 0
      (add1 (hukairs (cdr l)))))
```

Yes, that's better.

But what happened to the function?

It is no longer recursive.

And what does it do?

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

And what does *huhairs* do?

Who cares. The function doesn't work for non-empty lists in any case.

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

We think $length_0$ is great because the function only measures lists of length 0.

How would you write a function that measures the length of lists that contain one item?

Well, we could try the following.

```
(lambda (l)
  (if (null? l) 0
      (add1 (length0 (cdr l)))))
```

Almost, but (**define** ......) doesn't work for *length*$_0$.

So? Replace *length*$_0$ by its definition.

```
(lambda (l)
  (if (null? l) 0
      (add1
       ((lambda (l)
          (if (null? l) 0
              (add1 (hukairs (cdr l)))))
        (cdr l)))))
```

And what's a good name for this function?

That's easy: *length*$_1$.

Is this the function that would measure the lenght of lists that contain two items?

```
(lambda (l)
  (if (null? l) 0
      (add1
       ((lambda (l)
          (if (null? l) 0
              (add1
               ((lambda (l)
                  (if (null? l) 0
                      (add1
                       (hukairs
                        (cdr l)))))
                (cdr l)))))
        (cdr l)))))
```

Yes, this is *length*$_2$. We just expand the call to *hukairs* to get the next version of *length*.

Now, what do you think recursion is?

What do you mean?

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

If we could write an *infinite* function, we could write $length_{\infty}$.

But we can't write an infinite function.

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

All these programs contain a function that looks like *length*, and that's not right.

No, let's *abstract* out these patterns.

Is this the right way to rewrite $length_0$ so that *length* reappears?

```
((lambda (length)
   (lambda (l)
     (if (null? l) 0
        (add1 (length (cdr l))))))
 hukairs)
```

It's worth a try.

Rewrite $length_1$ in the same style.

```
((lambda (length)
   (lambda (l)
     (if (null? l) 0
        (add1 (length (cdr l))))))
 ((lambda (length)
    (lambda (l)
      (if (null? l) 0
         (add1 (length (cdr l))))))
  hukairs))
```

And $length_2$.

```
((lambda (length)
   (lambda (l)
     (if (null? l) 0
         (add1 (length (cdr l))))))
 ((lambda (length)
    (lambda (l)
      (if (null? l) 0
          (add1 (length (cdr l))))))
  ((lambda (length)
     (lambda (l)
       (if (null? l) 0
           (add1 (length (cdr l))))))
   hukairs)))
```

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?

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

Ok, do this to $length_0$.

No problem.

```
((lambda (mk-length)
   (mk-length hukairs))
 (lambda (length)
   (lambda (l)
     (if (null? l) 0
         (add1 (length (cdr l)))))))
```

Is this $length_1$?

```
((lambda (mk-length)
   (mk-length
    (mk-length hukairs)))
 (lambda (length)
   (lambda (l)
     (if (null? l) 0
         (add1 (length (cdr l)))))))
```

It sure is. And this is $length_2$.

```
((lambda (mk-length)
   (mk-length
    (mk-length
     (mk-length hukairs))))
 (lambda (length)
   (lambda (l)
     (if (null? l) 0
         (add1 (length (cdr l)))))))
```

Can you do $length_3$?

Here we go.

```
((lambda (mk-length)
   (mk-length
    (mk-length
     (mk-length
      (mk-length hukairs)))))
 (lambda (length)
   (lambda (l)
     (if (null? l) 0
         (add1 (length (cdr l)))))))
```

So what is recursion?

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 *hukairs* that is passed to the first *mk-length*.

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

That would 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 *huhairs* and the result of this on the *cdr* so that we get one more piece of the tower.

Then this is still $length_0$?

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

Yes. And when we apply *mk-length* once, we get $length_1$.

```
((lambda (mk-length)
   (mk-length mk-length))
 (lambda (mk-length)
   (lambda (l)
     (if (null? l) 0
         (add1 ((mk-length hukairs)
                (cdr l)))))))
```

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)
     (if (null? l) 0
         (add1 ((mk-length mk-length)
                (cdr l)))))))
```

It is *length*, of course.

Except that it no longer contains the function that looks like *length*. Can we fix that?

We could extract the self-application of *mk-length* and call it *length*.

Why?

Because it really makes the function *length*.

But (*mk-length mk-length*) is not a function. It only returns a function.

So what do we do?

Make the self-application of *mk-length* into a function.

No problem, we just use the old trick of wrapping a **lambda** around the application. After all, the self-application does return a function!

Which function?

The function *length*. Remember?

Ok, do it!

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

Is this ok to do?

Yes. Think about it. We always did the reverse: When we knew what the argument to a function was, we proceeded with the function body and used the argument value whenever we saw the parameter name.

Can we extract the function 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)
     (if (null? l) 0
        (add1 (length (cdr l)))))))
```

Yes.

What did we actually get back?

We extracted the *old* 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. At least, when we rewrite it a little bit.

```
(define Y
   (lambda (le)
     ((lambda (f)
         (le (lambda (x) ((f f) x))))
      (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 one more time and you will.
Does $(Y\ Y)$ work, too?	And how it works!!!