

CM20167 Topic 7: Y, Y, Y.

Guy McCusker

1W2.1

Something amazing!

This lecture is about something amazing.

We know that computing gains most of its power from the ability to perform actions repeatedly: things like while-loops and recursion give us an enormous amount of expressive power.

Languages without recursion (or unbounded iteration, like while-loops) usually have much reduced power. People who are interested in theory of computation can *prove* things like that.

In this lecture we will show how to create recursion in Lisp without using recursion.

A recursive function

Here's a simple recursive function that we've seen several times.

```
(defun length (l)
  (if (null l)
      0
      (+ 1 (length (cdr l))))
  )
)
```

The fact that we have recursion here means that we can use this function to measure a list of *any* length.

Only details of implementation stand between this function and the ability to measure arbitrarily long lists.

A non-recursive, non-useful function

```
(defun length0 (l)
  (if (null l)
      0
      (+ 1 (die (cdr l)))
  )
)
```

where die is some kind of failure. (Theoretically-minded people would put an infinite loop here.)

This function looks a lot like length, but it is only capable of measuring lists of length zero. Given a longer list, it dies.

A non-recursive, slightly more useful function

```
(defun length1 (l)
  (if (null l)
      0
      (+ 1 (length0 (cdr l)))
  )
)
```

This one will measure a list of length zero, or a list of length 1. Given a list of length two or more, it dies.

A pattern

If length- n is a function that can measure lists up to length n , then we can get length- n plus1, which works for list of length up to $n + 1$, as follows.

```
(defun length-nplus1 (l)
  (if (null l)
      0
      (+ 1 (length-n (cdr l)))
  )
)
```

Turning the pattern into a function

As we did with `foldr`, when a pattern comes up, it can be fruitful to turn it into a higher-order function.

Here's a function which will turn `length-n` into `length-nplus1`:

```
(defun make-length (f)
  (lambda (l)
    (if (null l)
        0
        (+ 1 (f (cdr l))))
    )
  )
)
```

We can now make a length function

Each application of `make-length` lets us measure lists one longer than we could before. So:

- ▶ `(make-length die)` is `length0`
- ▶ `(make-length (make-length die))` is `length1`
- ▶ ...
- ▶ `(make-length (make-length (make-length ...)))`, going on forever, is `length`.

Obviously the last one does not really exist.

We need a fixed point

The `length` function is a *fixed-point* of the `make-length` function:

- ▶ `length` can measure lists of any length
- ▶ `make-length` turns any partial length-function into something that can measure lists that are one element longer
- ▶ So `make-length` turns `length` into a function which can measure lists of any length, or one longer...
- ▶ That is, `(make-length length)` is the same as `length`.

What this means is that `length` is a *fixed-point* of `make-length`: a value which `make-length` leaves unchanged.

Fixed points and infinite unwindings

If `(make-length length)` is the same as `length`, then

- ▶ it's also the same as `(make-length (make-length length))`
- ▶ so it's the same as `(make-length (make-length (make-length length)))`
- ▶ so it's the same as `(make-length (make-length (make-length (make-length length))))`
- ▶ ...
- ▶ so it's the same as `(make-length (make-length (make-length (make-length ...))))`, going on forever.

A sneaky move

The `make-length` function we just wrote takes as its argument a partial length function, that is, a function which behaves a lot like a length function but gives up eventually.

For the first and most useless partial length function,

```
(make-length die)
```

it doesn't really matter what `die` is: the resulting function successfully measures lists of length zero, and goes wrong otherwise.

So we could put anything we like in place of `die`.

The only other function we've got is `make-length` itself, so let's try that.

Self-application

- ▶ `(make-length make-length)` measures lists of length zero, and causes an error on longer lists
- ▶ `(make-length (make-length make-length))` measures lists of length one, and causes an error on longer lists
- ▶ `(make-length (make-length (make-length make-length)))` measures lists of length two, and causes an error on longer lists
- ▶ ...
- ▶ `(make-length (make-length (make-length ...)))`, going on forever, is `length`.

(make-length make-length)

The function that gets this all going is (make-length make-length).
Since (make-length f) is:

```
(lambda (l)
  (if (null l)
      0
      (+ 1 (f (cdr l))))
  )
)
```

it must be the case that (make-length make-length) is

```
(lambda (l)
  (if (null l)
      0
      (+ 1 (make-length (cdr l))))
  )
)
```

Nearly recursive looking

Since (make-length make-length) is

```
(lambda (l)
  (if (null l)
      0
      (+ 1 (make-length (cdr l))))
  )
)
```

we *almost* have a function that calls itself. The only thing we need to change is to turn

```
(make-length (cdr l))
```

into

```
((make-length make-length) (cdr l))
```

Can we do that?

Yes we can!

Let's change make-length so that (make-length make-length) will end up calling itself:

```
(defun make-length (f)
  (lambda (l)
    (if (null l)
        0
        (+ 1 ((f f) (cdr l))))
    )
  )
)
```

Self-application again

With this definition, `(make-length make-length)` will end up calling itself: it's the function

```
(lambda (l)
  (if (null l)
      0
      (+ 1 ((make-length make-length) (cdr l)))
  )
)
```

But does it work?

Let's find out.

```
user> ((make-length make-length) ())
```

```
0
```

```
user> ((make-length make-length) '(a b c d e f g h i j))
```

```
10
```

Holy mackerel! It does work!

We've defined a recursive function without any recursion at all!

The key to it is *self-application*: applying a function to itself. This is not possible in lots of languages, but it is in Lisp.

Doing this trick in general

We can perform the same trick to define any recursive function of one argument via self-application.

First, notice that the sneakiest step was turning the `make-length` function into something with a self-application in it:

```
(defun make-length (f)
  (lambda (l)
    (if (null l)
        0
        (+ 1 (f (cdr l)))
    )
  )
)

(defun make-length (f)
  (lambda (l)
    (if (null l)
        0
        (+ 1 ((f f) (cdr l)))
    )
  )
)

becomes
```

Introducing self-application

We could have introduced self-application in a roundabout way like this: if `make-length` is the original, non-self-applicative version, then the self-applicative version is given by

```
(lambda (f) (make-length (lambda (a) ((f f) a))))
```

You may need to think about this for a while, but it is true:

```
user> (setq mad (lambda (f) (make-length (lambda (a) ((f f) a)))))
#<Procedure #1d3590>
user> ((mad mad) ())
0
user> ((mad mad) '(1 2 3 4))
4
```

Making the maker make itself

Having created the self-applicative maker, we can create the “recursive” function by applying it to itself, as before.

One way to write that is:

```
((lambda (f) (f f))
 (lambda (f) (make-length (lambda (a) ((f f) a)))))
)
```

Let's try it out.

This must be madness

```
user> (setq madness ((lambda (f) (f f))
 (lambda (f) (make-length (lambda (a) ((f f) a)))))
)
#<Procedure #1d52b8>
user> (madness ())
0
user> (madness '(1 2 3 4))
4
```

Abstracting the maker

We can now create a function to turn a maker into a recursion. It's traditionally called "Y". Why? Why not.

```
(defun Y (maker)
  ((lambda (f) (f f))
   (lambda (f) (maker (lambda (a) ((f f) a)))))
  )
)
```

```
user> (setq mylen (Y make-length))
#<Procedure #1d52b8>
user> (mylen ())
0
user> (mylen '(1 2 3 4 5))
5
```

Meet thy maker

Let's see how to do this to create arbitrary recursive functions. Given a definition like this:

```
(defun myfunction (a)
  body-of-function-that-may-call-myfunction-recursively
)
```

we create a "maker" function:

```
(defun mymaker (f)
  (lambda (a)
    body-of-myfunction-but-with-f-instead-of-recursive-call
  )
)
```

Y does the job

Now (Y mymaker) is the recursive function we were looking for, but with no recursion at all!

Yes, it's weird.

Example: quicksort

Quicksort is:

```
(defun qsort (l)
  (if (null l)
      nil
      (concatenate
       (qsort (filter (lambda (a) (< a (car l))) (cdr l)))
       (list (car l))
       (qsort (filter (lambda (a) (not (< a (car l))) (cdr l)))
              )
       )
  )
)
```

Example: quicksort

So its maker is:

```
(defun qsortmaker (f)
  (lambda (l)
    (if (null l)
        nil
        (concatenate
         (f (filter (lambda (a) (< a (car l))) (cdr l)))
         (list (car l))
         (f (filter (lambda (a) (not (< a (car l))) (cdr l)))
            )
         )
    )
  )
)
```

and now (Y qsortmaker) is a sorting function!

