What is Recursion???

Regarding Chapter 9 of
The Little Schemer
We are used to thinking of recursion as "a function calling itself", but what does that mean? We can make a function with a lambda expression. For example, \((\text{lambda} (x) (* 2 x))\) is a function that takes a number and doubles it.

We can create such a function without defining it; \textit{define} extends the environment and that shouldn't have anything to do with creating a function. But what about a function such as \texttt{length}:

\[
\begin{array}{l}
\text{(define length}
\end{array}
\]

\[
\begin{array}{l}
\quad \text{(lambda (lat)}
\end{array}
\]

\[
\begin{array}{l}
\quad\quad \text{(cond}
\end{array}
\]

\[
\begin{array}{l}
\quad\quad\quad \text{[(null? lat) 0]}
\end{array}
\]

\[
\begin{array}{l}
\quad\quad\quad \text{[else (add1 (length (cdr lat))))]]})
\end{array}
\]

How can we define such a function without extending the environment?
Well, first of all we need some place to start. I am going to extend the environment with one special function, which eventually we won't need.

(define eternity (lambda (x) (eternity x)))

This is the stupid recursion that your mother warned you about. For any argument a, (eternity a) never terminates.
Now consider the following function:

\[
\text{(define L (lambda (f) (lambda (lat) (cond [(null? lat) 0] [else (add1 (f (cdr lat)))])))))}
\]

The body here is somewhat like our length function.
(define L (lambda (f)
    (lambda (lat)
        (cond
            [(null? lat) 0]
            [else (add1 (f (cdr lat)))]))))

Consider
  (define L0 (L eternity))
This is
  (lambda (lat)
      (cond
          [(null? lat) 0]
          [else (add1 (eternity (cdr lat)))]))

This returns 0 for an empty lat, and fails if the lat is not empty.
(define L (lambda (f)
  (lambda (lat)
    (cond
     [(null? lat) 0]
     [else (add1 (f (cdr lat)))])))

(define L0 (L eternity))

Now consider
  (define L1 (L L0)) which is (L (L eternity)).
This is (lambda (lat)
  (cond
    [(null? lat) 0]
    [else (add1 (L0 (cdr lat)))]))
This returns 0 if the lat is empty, 1 if the (cdr lat) is empty, and fails otherwise.
(define L  (lambda (f)
  (lambda (lat)
    (cond
      [(null? lat) 0]
      [else (add1 (f (cdr lat)))]))))

(define L0  (L eternity))
(define L1 (L L0)) which is (L (L eternity)).

Similarly we can define
  (define L2 (L L1)) which is (L (L (L eternity)))
  (define L3 (L L2)) and so forth.

(L3 lat) correctly gives the length of any lat with length 3 or less, but fails for lats longer than 3.
(define L (lambda (f)
    (lambda (lat)
        (cond
            [(null? lat) 0]
            [else (add1 (f (cdr lat)))]))))

Here is another way to get at this. Consider

(define M1 ((lambda (x) (x x))
    (lambda (f)
        (lambda (lat)
            (lambda (lat)
                (cond
                    [(null? lat) 0]
                    [else (add1 ((f eternity) (cdr lat)))]))))))

It is hard to even parse this. M1 is the application of
(lambda (x) (x x)) to the lambda (f) expression.
(define M1 ((lambda (x) (x x))
  (lambda (f)
    (lambda (lat)
      (cond
        [(null? lat) 0]
        [else (add1 ((f eternity) (cdr lat)))]))))

If we let x be that lambda(f) expression, it isn't hard to see that (x eternity) is exactly the same as (L eternity), which we called L0, and (x x) is

  (lambda (lat)
    (cond
      [(null? lat) 0]
      [else (add1 (L0 (cdr lat)))]))

which is L1. In other words, M1 is the same as L1.
(define M1 ((lambda (x) (x x))
         (lambda (f)
         (lambda (lat)
         (cond
         [(null? lat) 0]
         [else (add1 ((f eternity) (cdr lat))))]))))

Finally, instead of falling back on eternity to fail if we go too far
through the list, apply f back to itself:
(define N ((lambda (x) (x x))
         (lambda (f)
         (lambda (lat)
         (cond
         [(null? lat) 0]
         [else (add1 ((f f) (cdr lat))))]))))
(define N ((lambda (x) (x x))
    (lambda (f)
        (lambda (lat)
            (cond
                [(null? lat) 0]
                [else (add1 ((f f) (cdr lat)))]))))

This is easier to think about if we let X be that lambda(f) expression. M is (X X), which is
    (lambda (lat)
        (cond
            [(null? lat) 0]
            [(else (add1 ((X X) (cdr lat)))]))

And this is exactly the length function!
Don't let the defines we have done fool you; we are saying that the length function is the result of applying

\[(\lambda (f) \\
  (\lambda (lat) \\
    (\text{cond} \\
      ((\text{null?} \ lat) 0) \\
      [\text{else} (\text{add1} ((f f) (\text{cdr} lat)))])])\]

to itself. We are getting recursion here without extending the environment. No function is "calling itself", yet we are recursing through the lat.
To recap, if we let $X$ be
\[
\lambda (f) \\
\quad \lambda (\text{lat}) \\
\quad \text{(cond} \\
\quad \quad \quad \text{((null? \text{lat}) 0]} \\
\quad \quad \quad \text{[else (add1 ((f f) (cdr \text{lat))][(]]))))
\]

then $(X \ X)$ is the length function for lats. We can write other recursions in this style.
\[
\text{(define Y (lambda (f);} \\
\quad \text{(lambda (a \text{lat})} \\
\quad \quad \text{(cond} \\
\quad \quad \quad \text{((null? \text{lat}) #f]} \\
\quad \quad \quad \text{[(eq? a (car \text{lat})) #t]} \\
\quad \quad \quad \text{[else ( (f f) a (cdr \text{lat)))]]}))
\]

Then $(Y \ Y)$ is the member? function.
(define Z (lambda (f)
    (lambda (a lat)
      (cond
        [(null? lat) null]
        [(eq? a (car lat)) (cadr lat)]
        [else (cons (car lat) ( (f f) a (cadr lat)))])))))

(Z Z) is the rember function
We can recurse on numbers as easily as lats:

\[
\begin{align*}
(\text{define } W &= (\lambda (f) \times 2)
    \\
    &\times 3
    \\
    &\times 4
    \\
    &\times 5)
\end{align*}
\]

\[
(W W) \text{ is the factorial function. You knew we couldn't do recursion and not mention factorials.}
\]
In Chapter 9 of *The Little Schemer* Friedman and Felleisen take this one step further.

Remember that we produced the length function as

```
(define N ((lambda (x) (x x))
  (lambda (f)
    (lambda (lat)
      (lambda (cond
        ((null? lat) 0)
      [else (add1 ((f f) (cdr lat)))]))))))
```

They rewrite this so it is the result of applying a complex function $Y$ to

```
(lambda (length)
  (lambda (lat)
    (cond
      ((null? lat) 0)
    [else (add1 (length (cdr lat)))])))
```

This actually looks like the usual definition of length, surrounded by a lambda `(length)`.

That complex function $Y$ is called the "Y Combinator". It was discovered by Haskell Curry.