In order to understand unbounded recursion, one must understand unbounded recursion.

Recursion is nameless

The implementation of recursion in LETREC is whack at worst and whack at best.

Here's factorial, for example :

```
((fix (lambda (self n)
                (if (zero? n)
                    1
                    (* n (self self (- n 1))))))
      5)
; evaluates to 120
```

For starters, LETREC allows recursive definitions without names. Secondly, what on earth is this `fix` form? Am I admitting to the fact that the way I've implemented recursion requires `fixing`? Also, what is this `self`? How can a procedure take *itself* as it's parameter? That makes no sense!

Recursion as an equation

In order to justify my case of a very whacky implementation of recursion (the honest justification is laziness but using abstract mathematics makes it look otherwise) I will start with the classic example of a while loop.

Let us say we have a programming language like follows :

```
bool-expr ::= true | false | bool and bool | bool or bool | not bool
int-expr ::= integer | int-expr + int-expr | int-expr * int-expr | int-expr - int-expr
expr ::= bool-expr | int-expr
assignment ::= variable <- expr
command ::= skip | assignment | if bool-expr then command else command
                 | while bool-expr do command | command ; command
```

Basically integers, booleans, if-else, assignments and while loops and a `skip` command that skips over to the next operation. (think empty semicolon `;` in langauge like C++)

Observe that `while` loops would, then, satisfy the following equation :

```
while B do C = if B then { C ; while B do C }
               else skip
```

Parametrising over `B` (`B` ranges over `bool-expr`) and `C` (ranges over `command`) we get that the `while` form $W_{B,C}$ is of the form

$$W_{B,C} = \phi(W_{B,C})$$

where ϕ takes any form `f` to the `if B then C ; f else skip`.

More formally, `while` is completely defined by the above equation and we say that $W_{B,C}$ is the *fixpoint* of ϕ. Such equations are called *fixpoint equations*.

We have an entire field of mathematics dedicated to the study of fixpoints - domain theory. It should be no surprise that since recursion is *defined* mathematically using fixpoint equations; domain theory is a favourite subject of computer scientists.

Recursion is defined recursively

Let us go to the definition of factorial in a classical programming language like Python :

```
def fact(n):
  if n == 0:
    return 1
  else:
    return n * fact(n - 1)
```

Observe that in this definition is implicit the fact that for `fact(n)` to be well-defined we need a proof of termination of `fact(n - 1)`. It should also be evident that while not explicitly parametrised over some form of `self`, this definition still implicitly adds `fact` to the environment in which it is defined.

Any implementation of a programming langauge is inevitably written in *some* langauge (the metalanguage) and practically every metalanguage will have recursion. This allows the implementor to exploit the recursion already present in the metalanguage to define the operational semantics of recursion in the object language.

Recursive functions as fixpoints?

If any recursive definition can be written as a fixpoint equation and doing so allows us to talk about recursion without circular definitions - the natural question to ask is what about recursive functions? The ubiquitous form of recursion must also be described by this.

In order to see the sort of equation that defines recursive functions, let us make a few things more explicit in our older factorial function :

```
def fact(self, n):
  if n == 0 :
    return 1
  else :
    return n * self(self, n - 1)
```

Observe that if `self` is only capable of computing the factorial of $n-1$ that too is good enough - we can compute the factorial of n through that. So what we have established is that given a function that can compute the factorial of $n-1$ it is possible to construct a function that can compute the factorial of n.

This is starting to sound suspiciously similar to induction!

We need a higher-order function - one that takes an approximation of factorial that can compute upto $n!$ and returns a new function that can compute $(n+1)!$.

This leads to a certain strategy - recursive functions must be fixpoints of *higher-order functions*.

Let us try to actually write down such a function :

```
def fact_combinator(approx, n):
  def base_case(input):
    return 1
  def inductive_case(input):
    if input == n :
      return n * approx(n - 1)
    return approx(input)
  if n == 0:
    return base_case
  return inductive_case
```

If given the nth approximation to factorial, this function will return the $(n+1)$th approximation.

What about our *first ever approximation*? Since this procedure is never invoked we can put it as anything : even a function that doesn't even terminate! Indeed, the "zeroth" approximation is chosen as the "empty function" - the one that doesn't return for any input whatsoever :

```
def empty(n):
  return empty(n)
```

Observe that using the first ever approximation is a very general trick that will work for *any fixpoint equation* rather than just the one for factorials!

Finding fixpoints - the mathematics

Technical Jargon

The only real theorem we need is that for any given bounded partial order, any monotone sequence converges to a certain limit that can be calculated using the join or the meet operation as appropriate.

I'm going to be intentionally informal here and that probably sounded like Greek.

A partial order is essentially a generalisation of the familiar $\leq$ operator for abstract mathematical objects rather than just numbers. An important restriction is that for any two a and b it is not necessary that $a \leq b$ or $b \leq a$.

A good example is the | "divides" operation on natural numbers. Every number is divisible by itself. If a is divisible by b and b is divisible by c then a is divisible by c. Note that for any two a and b it is not necessary that a divides b or that b divides a - it might be the case that neither is true.

Another good example from computer science is the topological ordering of a graph.

What's a *bounded* partial order? Essentially, whatever objects we are considering have a "maximum" and a "minimum" in the sense that $\exists a.\forall b.a \leq b$ and $existsa.\forall b.b \leq a$.

Now, what's a *monotone sequence*? Well any sequence a_n (finite or infinite) such that either $a_i \leq a_{i+1}$ or $a_{i+1} \leq a_i$.

What we're saying is that *any* such sequence always *converges*. Of course, I have not actually defined convergence. One way of thinking about it is that there is one *unique* element that to which the elements in our sequence appear to get closer to. For reasons of brevity I will not go too much into the discussion about convergence and instead state the result and we'll accept it as an axiom. Also, since we will only be talking a particular kind of partial order - the partial order of subsets of a set, I will only talk about that.

Observe that given two subsets of some set U, say A and B, the relation $\subseteq$ is a partial order!

Also, the empty set is the subset of any subset of U and the subset U is the superset of any subset of U. It is not *just* a partial order - it is a *bounded partial order*!

Now we will state the theorems in their entire glory.

Assume that F_n is a monotone sequence of subsets of some universe U. Specifically, $F_{n+1} \leq F_n$.

Then we define the *limit* of this sequence as the countable union of it's elements :

$$\lim_{n\to\infty} F_n = \bigcup_{i=1}^{\infty} F_i$$

This will suffice for our purposes.

Sequence of approximations