PL: Lecture #12

Tuesday, October 15th

Recursion without the Magic

> 👉 ***PLAI §22.4*** *(we go much deeper)*

> *Note: This explanation is similar to the one you can find in "The Why of Y", by Richard Gabriel.*

To implement recursion without the `define` magic, we first make an observation: this problem does *not* come up in a dynamically-scoped language. Consider the `let`-version of the problem:

```
#lang pl dynamic
(let ([fact (lambda (n)
              (if (zero? n) 1 (* n (fact (- n 1)))))])
  (fact 5))
```

This works fine — because by the time we get to evaluate the body of the function, `fact` is already bound to itself in the current dynamic scope. (This is another reason why dynamic scope is perceived as a convenient approach in new languages.)

Regardless, the problem that we have with lexical scope is still there, but the way things work in a dynamic scope suggest a solution that we can use now. Just like in the dynamic scope case, when `fact` is called, it does have a value — the only problem is that this value is inaccessible in the lexical scope of its body.

Instead of trying to get the value in via lexical scope, we can imitate what happens in the dynamically scoped language by passing the `fact` value to itself so it can call itself (going back to the original code in the broken-scope language):

```
(define (fact self n)  ;***
  (if (zero? n) 1 (* n (self (- n 1)))))
(fact fact 5)          ;***
```

except that now the recursive call should still send itself along:

```
(define (fact self n)
  (if (zero? n) 1 (* n (self self (- n 1))))) ;***
(fact fact 5)
```

The problem is that this required rewriting calls to `fact` — both outside and recursive calls inside. To make this an acceptable solution, calls from both places should not change. Eventually, we should be able to get a working `fact` definition that uses just

```
(lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))
```

The first step in resolving this problem is to curry the `fact` definition.

```
(define (fact self)                   ;***
  (lambda (n)                         ;***
    (if (zero? n)
      1
      (* n ((self self) (- n 1))))))  ;***
((fact fact) 5)                       ;***
```

Now `fact` is no longer our factorial function — it's a function that constructs it. So call it `make-fact`, and bind `fact` to the actual factorial function.

```
(define (make-fact self)              ;***
  (lambda (n)
    (if (zero? n) 1 (* n ((self self) (- n 1))))))
(define fact (make-fact make-fact))   ;***
(fact 5)                              ;***
```

We can try to do the same thing in the body of the factorial function: instead of calling `(self self)`, just bind `fact` to it:

```
(define (make-fact self)
  (lambda (n)
    (let ([fact (self self)])         ;***
      (if (zero? n)
        1
        (* n (fact (- n 1)))))))      ;***
(define fact (make-fact make-fact))
(fact 5)
```

This works fine, but if we consider our original goal, we need to get that local `fact` binding outside of the `(lambda (n) ...)` — so we're left with a definition that uses the factorial expression as is. So, swap the two lines:

```
(define (make-fact self)
  (let ([fact (self self)])  ;***
    (lambda (n)              ;***
      (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact (make-fact make-fact))
(fact 5)
```

But the problem is that this gets us into an infinite loop because we're trying to evaluate `(self self)` too ea(ge)rly. In fact, if we ignore the body of the `let` and other details, we basically do this:

```
(define (make-fact self) (self self)) (make-fact make-fact)
--reduce-sugar-->
(define make-fact (lambda (self) (self self))) (make-fact make-fact)
--replace-definition-->
((lambda (self) (self self)) (lambda (self) (self self)))
--rename-identifiers-->
((lambda (x) (x x)) (lambda (x) (x x)))
```

And this expression has an interesting property: it reduces to itself, so evaluating it gets stuck in an infinite loop.

So how do we solve this? Well, we know that `(self self)` *should* be the same value that is the factorial function itself — so it must be a one-argument function. If it's such a function, we can use a value that is equivalent, except that it will not get evaluated until it is needed, when the function is called. The trick here is the observation that `(lambda (n) (add1 n))` is really the same as `add1` (provided that `add1` is a one-argument function), except that the `add1` part doesn't get evaluated until the function is called. Applying this trick to our code produces a version that does not get stuck in the same infinite loop:

```
(define (make-fact self)
  (let ([fact (lambda (n) ((self self) n))]) ;***
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact (make-fact make-fact))
(fact 5)
```

Continuing from here — we know that

```
(let ([x v]) e)  is the same as  ((lambda (x) e) v)
```

(remember how we derived `fun` from a `with`), so we can turn that `let` into the equivalent function application form:

```
(define (make-fact self)
  ((lambda (fact)                      ;***
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
   (lambda (n) ((self self) n))))      ;***
(define fact (make-fact make-fact))
(fact 5)
```

And note now that the (lambda (fact) …) expression is everything that we need for a recursive definition of `fact` — it has the proper factorial body with a plain recursive call. It's almost like the usual value that we'd want to define `fact` as, except that we still have to abstract on the recursive value itself. So lets move this code into a separate definition for `fact-step`:

```
(define fact-step                      ;***
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define (make-fact self)
  (fact-step                           ;***
   (lambda (n) ((self self) n))))
(define fact (make-fact make-fact))
(fact 5)
```

We can now proceed by moving the `(make-fact make-fact)` self application into its own function which is what creates the real factorial:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define (make-fact self)
  (fact-step
   (lambda (n) ((self self) n))))
(define (make-real-fact) (make-fact make-fact)) ;***
(define fact (make-real-fact))                  ;***
(fact 5)
```

Rewrite the `make-fact` definition using an explicit `lambda`:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define make-fact                      ;***
  (lambda (self)                       ;***
    (fact-step
     (lambda (n) ((self self) n)))))
(define (make-real-fact) (make-fact make-fact))
(define fact (make-real-fact))
(fact 5)
```

and fold the functionality of `make-fact` and `make-real-fact` into a single `make-fact` function by just using the value of `make-fact` explicitly instead of through a definition:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define (make-real-fact)
  (let ([make (lambda (self)                    ;***
```

```
                (fact-step                          ;***
                 (lambda (n) ((self self) n)))))]) ;***
    (make make)))
(define fact (make-real-fact))
(fact 5)
```

We can now observe that `make-real-fact` has nothing that is specific to factorial — we can make it take a “core function” as an argument:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define (make-real-fact core)                       ;***
  (let ([make (lambda (self)
                (core                               ;***
                 (lambda (n) ((self self) n)))))])
    (make make)))
(define fact (make-real-fact fact-step))            ;***
(fact 5)
```

and call it `make-recursive`:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define (make-recursive core)                ;***
  (let ([make (lambda (self)
                (core
                 (lambda (n) ((self self) n)))))])
    (make make)))
(define fact (make-recursive fact-step))  ;***
(fact 5)
```

We’re almost done now — there’s no real need for a separate `fact-step` definition, just use the value for the definition of `fact`:

```
(define (make-recursive core)
  (let ([make (lambda (self)
                (core
                 (lambda (n) ((self self) n)))))])
    (make make)))
(define fact
  (make-recursive
   (lambda (fact)                                          ;***
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))) ;***
(fact 5)
```

turn the `let` into a function form:

```
(define (make-recursive core)
  ((lambda (make) (make make))                ;***
   (lambda (self)                             ;***
     (core (lambda (n) ((self self) n))))))  ;***
(define fact
  (make-recursive
   (lambda (fact)
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
(fact 5)
```

do some renamings to make things simpler — `make` and `self` turn to `x`, and `core` to `f`:

```
(define (make-recursive f)                    ;***
  ((lambda (x) (x x))                         ;***
   (lambda (x) (f (lambda (n) ((x x) n))))) ;***
(define fact
  (make-recursive
   (lambda (fact)
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
(fact 5)
```

or we can manually expand that first (lambda (x) (x x)) application to make the symmetry more obvious (not really surprising because it started with a `let` whose purpose was to do a self-application):

```
(define (make-recursive f)
  ((lambda (x) (f (lambda (n) ((x x) n))))   ;***
   (lambda (x) (f (lambda (n) ((x x) n))))) ;***
(define fact
  (make-recursive
   (lambda (fact)
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
(fact 5)
```

And we finally got what we were looking for: a general way to define *any* recursive function without any magical `define` tricks. This also work for other recursive functions:

```
#lang pl broken
(define (make-recursive f)
  ((lambda (x) (f (lambda (n) ((x x) n))))
   (lambda (x) (f (lambda (n) ((x x) n))))))
(define fact
  (make-recursive
   (lambda (fact)
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
(fact 5)
(define fib
  (make-recursive
   (lambda (fib)
     (lambda (n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))))))
(fib 8)
(define length
  (make-recursive
   (lambda (length)
     (lambda (l) (if (null? l) 0 (+ (length (rest l)) 1))))))
(length '(x y z))
```

A convenient tool that people often use on paper is to perform a kind of a syntactic abstraction: “assume that whenever I write (twice foo) I really meant to write (foo foo)”. This can often be done as plain abstractions (that is, using functions), but in some cases — for example, if we want to abstract over definitions — we just want such a rewrite rule. (More on this towards the end of the course.) The broken-scope language does provide such a tool — `rewrite` extends the language with a rewrite rule. Using this, and our `make-recursive`, we can make up a recursive definition form:

```
(rewrite (define/rec (f x) E)
      => (define f (make-recursive (lambda (f) (lambda (x) E)))))
```

In other words, we’ve created our own “magical definition” form. The above code can now be written in almost the same way it is written in plain Racket:

```
#lang pl broken
(define (make-recursive f)
```

```
  ((lambda (x) (f (lambda (n) ((x x) n))))
   (lambda (x) (f (lambda (n) ((x x) n))))))
(rewrite (define/rec (f x) E)
      => (define f (make-recursive (lambda (f) (lambda (x) E)))))
;; examples
(define/rec (fact n) (if (zero? n) 1 (* n (fact (- n 1)))))
(fact 5)
(define/rec (fib n) (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2)))))
(fib 8)
(define/rec (length l) (if (null? l) 0 (+ (length (rest l)) 1)))
(length '(x y z))
```

Finally, note that make-recursive is limited to 1-argument functions only because of the protection from eager evaluation. In any case, it can be used in any way you want, for example,

```
(make-recursive (lambda (f) (lambda (x) f)))
```

is a function that *returns itself* rather than calling itself. Using the rewrite rule, this would be:

```
(define/rec (f x) f)
```

which is the same as:

```
(define (f x) f)
```

in plain Racket.

The Core of `make-recursive`

As in Racket, being able to express recursive functions is a fundamental property of the language. It means that we can have loops in our language, and that's the essence of making a language powerful enough to be TM-equivalent — able to express undecidable problems, where we don't know whether there is an answer or not.

The core of what makes this possible is the expression that we have seen in our derivation:

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

which reduces to itself, and therefore has no value: trying to evaluate it gets stuck in an infinite loop. (This expression is often called "Omega".)

This is the key for creating a loop — we use it to make recursion possible. Looking at our final `make-recursive` definition and ignoring for a moment the "protection" that we need against being stuck prematurely in an infinite loop:

```
(define (make-recursive f)
  ((lambda (x) (x x))
   (lambda (x) (f (x x)))))
```

we can see that this is almost the same as the Omega expression — the only difference is that application of `f`. Indeed, this expression (the result of (make-recursive F) for some `F`) reduces in a similar way to Omega:

```
((lambda (x) (x x)) (lambda (x) (F (x x))))
((lambda (x) (F (x x))) (lambda (x) (F (x x))))
(F ((lambda (x) (F (x x))) (lambda (x) (F (x x)))))
(F (F ((lambda (x) (F (x x))) (lambda (x) (F (x x))))))
(F (F (F ((lambda (x) (F (x x))) (lambda (x) (F (x x)))))))
...
```

which means that the actual value of this expression is:

```
(F (F (F ...forever...)))
```

This definition would be sufficient if we had a lazy language, but to get things working in a strict one we need to bring back the protection. This makes things a little different — if we use `(protect f)` to be a shorthand for the protection trick,

```
(rewrite (protect f) => (lambda (x) (f x)))
```

then we have:

```
(define (make-recursive f)
  ((lambda (x) (x x)) (lambda (x) (f (protect (x x))))))
```

which makes the (make-recursive F) evaluation reduce to

```
(F (protect (F (protect (F (protect (...forever...)))))))
```

and this is still the same result (as long as `F` is a single-argument function).

(Note that `protect` cannot be implemented as a plain function!)

Denotational Explanation of Recursion

> *Note: This explanation is similar to the one you can find in "The Little Schemer" called "(Y Y) Works!", by Dan Friedman and Matthias Felleisen.*

The explanation that we have now for how to derive the `make-recursive` definition is fine — after all, we did manage to get it working. But this explanation was done from a kind of an operational point of view: we knew a certain trick that can make things work and we pushed things around until we got it working like we wanted. Instead of doing this, we can re-approach the problem from a more declarative point of view.

So, start again from the same broken code that we had (using the broken-scope language):

```
(define fact
  (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))
```

This is as broken as it was when we started: the occurrence of `fact` in the body of the function is free, which means that this code is meaningless. To avoid the compilation error that we get when we run this code, we can substitute *anything* for that `fact` — it's even better to use a replacement that will lead to a runtime error:

```
(define fact
  (lambda (n) (if (zero? n) 1 (* n (777 (- n 1))))) ;***
```

This function will not work in a similar way to the original one — but there is one case where it *does* work: when the input value is `0` (since then we do not reach the bogus application). We note this by calling this function `fact0`:

```
(define fact0                  ;***
  (lambda (n) (if (zero? n) 1 (* n (777 (- n 1))))))
```

Now that we have this function defined, we can use it to write `fact1` which is the factorial function for arguments of `0` or `1`:

```
(define fact0
  (lambda (n) (if (zero? n) 1 (* n (777 (- n 1))))))
(define fact1
  (lambda (n) (if (zero? n) 1 (* n (fact0 (- n 1))))))
```

And remember that this is actually just shorthand for:

```
(define fact1
  (lambda (n)
    (if (zero? n)
      1
      (* n ((lambda (n)
              (if (zero? n)
                1
                (* n (777 (- n 1)))))
            (- n 1))))))
```

We can continue in this way and write `fact2` that will work for n<=2:

```
(define fact2
  (lambda (n) (if (zero? n) 1 (* n (fact1 (- n 1))))))
```

or, in full form:

```
(define fact2
  (lambda (n)
    (if (zero? n)
      1
      (* n ((lambda (n)
              (if (zero? n)
                1
                (* n ((lambda (n)
                        (if (zero? n)
                          1
                          (* n (777 (- n 1)))))
                      (- n 1)))))
            (- n 1))))))
```

If we continue this way, we *will* get the true factorial function, but the problem is that to handle *any* possible integer argument, it will have to be an infinite definition! Here is what it is supposed to look like:

```
(define fact0 (lambda (n) (if (zero? n) 1 (* n (777 (- n 1))))))
(define fact1 (lambda (n) (if (zero? n) 1 (* n (fact0 (- n 1))))))
(define fact2 (lambda (n) (if (zero? n) 1 (* n (fact1 (- n 1))))))
(define fact3 (lambda (n) (if (zero? n) 1 (* n (fact2 (- n 1))))))
...
```

The true factorial function is `fact-infinity`, with an infinite size. So, we're back at the original problem…

To help make things more concise, we can observe the repeated pattern in the above, and extract a function that abstracts this pattern. This function is the same as the `fact-step` that we have seen previously:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact0 (fact-step 777))
(define fact1 (fact-step fact0))
(define fact2 (fact-step fact1))
(define fact3 (fact-step fact2))
...
```

which is actually:

```
(define fact-step
  (lambda (fact)
```

```
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact0 (fact-step 777))
(define fact1 (fact-step (fact-step 777)))
(define fact2 (fact-step (fact-step (fact-step 777))))
...
(define fact
  (fact-step (fact-step (fact-step (... (fact-step 777) ...)))))
```

Do this a little differently — rewrite `fact0` as:

```
(define fact0
  ((lambda (mk) (mk 777))
   fact-step))
```

Similarly, `fact1` is written as:

```
(define fact1
  ((lambda (mk) (mk (mk 777)))
   fact-step))
```

and so on, until the real factorial, which is still infinite at this stage:

```
(define fact
  ((lambda (mk) (mk (mk (... (mk 777) ...))))
   fact-step))
```

Now, look at that `(lambda (mk) ...)` — it is an infinite expression, but for every actual application of the resulting factorial function we only need a finite number of `mk` applications. We can guess how many, and as soon as we hit an application of `777` we know that our guess is too small. So instead of `777`, we can try to use the maker function to create and use the next.

To make things more explicit, here is the expression that is our `fact0`, without the definition form:

```
((lambda (mk) (mk 777))
 fact-step)
```

This function has a very low guess — it works for 0, but with 1 it will run into the `777` application. At this point, we want to somehow invoke `mk` again to get the next level — and since `777` *does* get applied, we can just replace it with `mk`:

```
((lambda (mk) (mk mk))
 fact-step)
```

The resulting function works just the same for an input of `0` because it does not attempt a recursive call — but if we give it `1`, then instead of running into the error of applying `777`:

```
(* n (777 (- n 1)))
```

we get to apply `fact-step` there:

```
(* n (fact-step (- n 1)))
```

and this is still wrong, because `fact-step` expects a function as an input. To see what happens more clearly, write `fact-step` explicitly:

```
((lambda (mk) (mk mk))
 (lambda (fact)
   (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
```

The problem is in what we're going to pass into `fact-step` — its `fact` argument will not be the factorial function, but the `mk` function constructor. Renaming the `fact` argument as `mk` will make this more obvious (but not change the meaning):

```
((lambda (mk) (mk mk))
 (lambda (mk)
   (lambda (n) (if (zero? n) 1 (* n (mk (- n 1)))))))
```

It should now be obvious that this application of `mk` will not work, instead, we need to apply it on some function and *then* apply the result on `(- n 1)`. To get what we had before, we can use `777` as a bogus function:

```
((lambda (mk) (mk mk))
 (lambda (mk)
   (lambda (n) (if (zero? n) 1 (* n ((mk 777) (- n 1)))))))
```

This will allow one recursive call — so the definition works for both inputs of `0` and `1` — but not more. But that `777` is used as a maker function now, so instead, we can just use `mk` itself again:

```
((lambda (mk) (mk mk))
 (lambda (mk)
   (lambda (n) (if (zero? n) 1 (* n ((mk mk) (- n 1)))))))
```

And this is a *working* version of the real factorial function, so make it into a (non-magical) definition:

```
(define fact
  ((lambda (mk) (mk mk))
   (lambda (mk)
     (lambda (n) (if (zero? n) 1 (* n ((mk mk) (- n 1))))))))
```

But we're not done — we "broke" into the factorial code to insert that `(mk mk)` application — that's why we dragged in the actual value of `fact-step`. We now need to fix this. The expression on that last line

```
(lambda (n) (if (zero? n) 1 (* n ((mk mk) (- n 1)))))
```

is close enough — it is `(fact-step (mk mk))`. So we can now try to rewrite our `fact` as:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact
  ((lambda (mk) (mk mk))
   (lambda (mk) (fact-step (mk mk)))))
```

… and would fail in a familiar way! If it's not familiar enough, just rename all those `mk`s as `x`s:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact
  ((lambda (x) (x x))
   (lambda (x) (fact-step (x x)))))
```

We've run into the eagerness of our language again, as we did before. The solution is the same — the `(x x)` is the factorial function, so protect it as we did before, and we have a working version:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define fact
  ((lambda (x) (x x))
   (lambda (x) (fact-step (lambda (n) ((x x) n))))))
```

The rest should not be surprising now… Abstract the recursive making bit in a new `make-recursive` function:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
(define (make-recursive f)
  ((lambda (x) (x x))
   (lambda (x) (f (lambda (n) ((x x) n))))))
(define fact (make-recursive fact-step))
```

and now we can do the first reduction inside `make-recursive` and write the `fact-step` expression explicitly:

```
#lang pl broken
(define (make-recursive f)
  ((lambda (x) (f (lambda (n) ((x x) n))))
   (lambda (x) (f (lambda (n) ((x x) n))))))
(define fact
  (make-recursive
   (lambda (fact)
     (lambda (n) (if (zero? n) 1 (* n (fact (- n 1))))))))
```

and this is the same code we had before.

The Y Combinator

Our `make-recursive` function is usually called the *fixpoint operator* or the *Y combinator*.

It looks really simple when using the lazy version (remember: our version is the eager one):

```
(define Y
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
```

> *Note that if we do allow a recursive definition for Y itself, then the definition can follow the definition that we've seen:*
>
> ```
> (define (Y f) (f (Y f)))
> ```

And this all comes from the loop generated by:

```
((lambda (x) (x x)) (lambda (x) (x x)))
```

This expression, which is also called *Omega* (the `(lambda (x) (x x))` part by itself is usually called *omega* and then `(omega omega)` is *Omega*), is also the idea behind many deep mathematical facts. As an example for what it does, follow the next rule:

```
I will say the next sentence twice:
  "I will say the next sentence twice".
```

(Note the usage of colon for the first and quotes for the second — what is the equivalent of that in the lambda expression?)

By itself, this just gets you stuck in an infinite loop, as Omega does, and the Y combinator adds `F` to that to get an infinite chain of applications — which is similar to:

```
I will say the next sentence twice:
  "I will hop on one foot and then say the next sentence twice".
```

Sidenote: **see this SO question** and my answer, which came from the PLQ implementation.

The main property of Y

`fact-step` is a function that given any limited factorial, will generate a factorial that is good for one more integer input. Start with `777`, which is a factorial that is good for nothing (because it's not a function), and you can get `fact0` as

```
fact0 == (fact-step 777)
```

and that's a good factorial function only for an input of `0`. Use that with `fact-step` again, and you get

```
fact1 == (fact-step fact0) == (fact-step (fact-step 777))
```

which is the factorial function when you only look at input values of `0` or `1`. In a similar way

```
fact2 == (fact-step fact1)
```

is good for `0`…`2` — and we can continue as much as we want, except that we need to have an infinite number of applications — in the general case, we have:

```
fact-n == (fact-step (fact-step (fact-step ... 777)))
```

which is good for `0`…`n`. The *real* factorial would be the result of running `fact-step` on itself infinitely, it *is* `fact-infinity`. In other words (here `fact` is the *real* factorial):

```
fact = fact-infinity == (fact-step (fact-step ...infinitely...))
```

but note that since this is really infinity, then

```
fact = (fact-step (fact-step ...infinitely...))
     = (fact-step fact)
```

so we get an equation:

```
fact = (fact-step fact)
```

and a solution for this is going to be the real factorial. The solution is the *fixed-point* of the `fact-step` function, in the same sense that `0` is the fixed point of the `sin` function because

```
0 = (sin 0)
```

And the Y combinator does just that — it has this property:

```
(make-recursive f) = (f (make-recursive f))
```

or, using the more common name:

```
(Y f) = (f (Y f))
```

This property encapsulates the real magical power of Y. You can see how it works: since `(Y f) = (f (Y f))`, we can add an `f` application to both sides, giving us `(f (Y f)) = (f (f (Y f)))`, so we get:

```
(Y f) = (f (Y f)) = (f (f (Y f))) = (f (f (f (Y f)))) = ...
      = (f (f (f ...)))
```

and we can conclude that

```
(Y fact-step) = (fact-step (fact-step ...infinitely...))
              = fact
```

Yet another explanation for Y

Here's another explanation of how the Y combinator works. Remember that our `fact-step` function was actually a function that generates a factorial function based on some input, which is supposed to be the factorial function:

```
(define fact-step
  (lambda (fact)
    (lambda (n) (if (zero? n) 1 (* n (fact (- n 1)))))))
```

As we've seen, you can apply this function on a version of factorial that is good for inputs up to some n, and the result will be a factorial that is good for those values up to n+1. The question is *what is the fixpoint of `fact-step`*? And the answer is that if it maps fact_n factorial to fact_{n+1}, then the input will be equal to the output on the *infinitieth* `fact`, which is the *actual* factorial. Since Y is a fixpoint combinator, it gives us exactly that answer:

```
(define the-real-factorial (Y fact-step))
```

Typing the Y Combinator

Typing the Y combinator is a tricky issue. For example, in standard ML you must write a new type definition to do this:

```
datatype 'a t = T of 'a t -> 'a
val y = fn f => (fn (T x) => (f (fn a => x (T x) a)))
                 (T (fn (T x) => (f (fn a => x (T x) a))))
```

> *Can you find a pattern in the places where `T` is used? — Roughly speaking, that type definition is*
>
> ```
> ;; 't' is the type name, 'T' is the constructor (aka the variant)
> (define-type (RecTypeOf t)
>   [T ((RecTypeOf t) -> t)])
> ```
>
> *First note that the two `fn a => ...` parts are the same as our protection, so ignoring that we get:*
>
> ```
> val y = fn f => (fn (T x) => (f (x (T x))))
>                  (T (fn (T x) => (f (x (T x)))))
> ```
>
> *if you now replace `T` with `Quote`, things make more sense:*
>
> ```
> val y = fn f => (fn (Quote x) => (f (x (Quote x))))
>                  (Quote (fn (Quote x) => (f (x (Quote x)))))
> ```
>
> *and with our syntax, this would be:*
>
> ```
> (define (Y f)
>   ((lambda (qx)
>      (cases qx
>        [(Quote x) (f (x qx))]))
>    (Quote
>     (lambda (qx)
>       (cases qx
>         [(Quote x) (f (x qx))])))))
> ```
>
> *it's not really quotation — but the analogy should help: it uses `Quote` to distinguish functions as values that are applied (the `x`s) from functions that are passed as arguments.*

In OCaml, this looks a little different:

```
# type 'a t = T of ('a t -> 'a) ;;
type 'a t = T of ('a t -> 'a)
# let y f = (fun (T x) -> x (T x))
            (T (fun (T x) -> fun z -> f (x (T x)) z)) ;;
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# let fact = y (fun fact n -> if n < 1 then 1 else n * fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
- : int = 120
```

but OCaml has also a `-rectypes` command line argument, which will make it infer the type by itself:

```
# let y f = (fun x -> x x) (fun x -> fun z -> f (x x) z) ;;
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
# let fact = y (fun fact n -> if n < 1 then 1 else n * fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
- : int = 120
```

The translation of this to `#lang pl` is a little verbose because we don't have auto-currying, and because we need to declare input types to functions, but it's essentially a direct translation of the above:

```
(define-type (RecTypeOf t)
  [T ((RecTypeOf t) -> t)])
(: Y : (All (A B) ((A -> B) -> (A -> B)) -> (A -> B)))
(define (Y f)
  ((lambda ([x : (RecTypeOf (A -> B))])
     (cases x
       [(T x) (x (T x))]))
   (T (lambda ([x : (RecTypeOf (A -> B))])
        (cases x
          [(T x) (lambda ([z : A])
                   ((f (x (T x))) z))])))))
(define fact
  (Y (lambda ([fact : (Integer -> Integer)])
       (lambda ([n : Integer])
         (if (< n 1) 1 (* n (fact (sub1 n))))))))
(fact 5)
```

It is also possible to write this expression in "plain" Typed Racket, without a user-defined type — and we need to start with a proper type definition. First of all, the type of Y should be straightforward: it is a fixpoint operation, so it takes a `T -> T` function and produces its fixpoint. The fixpoint itself is some `T` (such that applying the function on it results in itself). So this gives us:

```
(: make-recursive : (T -> T) -> T)
```

However, in our case `make-recursive` computes a *functional* fixpoint, for unary `S -> T` functions, so we should narrow down the type

```
(: make-recursive : ((S -> T) -> (S -> T)) -> (S -> T))
```

Now, in the body of `make-recursive` we need to add a type for the `x` argument which is behaving in a weird way: it is used both as a function and as its own argument. (Remember — I will say the next sentence twice: "I will say the next sentence twice".) We need a recursive type definition helper (not a new type) for that:

```
(define-type (Tau S T) = (Rec this (this -> (S -> T))))
```

This type is tailored for our use of `x`: it is a type for a function that will *consume itself* (hence the `Rec`) and spit out the value that the `f` argument consumes — an `S -> T` function.

The resulting full version of the code:

```
(: make-recursive : (All (S T) ((S -> T) -> (S -> T)) -> (S -> T)))
(define-type (Tau S T) = (Rec this (this -> (S -> T))))
(define (make-recursive f)
  ((lambda ([x : (Tau S T)]) (f (lambda (z) ((x x) z))))
   (lambda ([x : (Tau S T)]) (f (lambda (z) ((x x) z))))))

(: fact : Number -> Number)
(define fact (make-recursive
              (lambda ([fact : (Number -> Number)])
                (lambda ([n : Number])
                  (if (zero? n)
                    1
                    (* n (fact (- n 1))))))))

(fact 5)
```