Topic. Lambda Calculus: Y-Combinator

Note. Recursive definitions of the factorial function

Implementation. Function with parameter

```
<rkt>
     (define (factorial n)
       (if (= n 0)
         1
         (* n (factorial (- n 1)))))
</rkt>
```

Implementation. Function with lambda

```
<rkt>
     (define factorial
       (λ(n)
         (if (= n 0)
           1
           (* n (factorial (sub1 n))))))

</rkt>
```

==

Note. What the Y combinator is and what it does

The Y combinator is a higher-order function. It takes a single argument, which is a function that isn't recursive. It returns a version of the function which is recursive. It gives us a way to get recursion in a language that supports first-class functions but that doesn't have recursion built in to it.

==

Note. Lazy or Strict Evaluation?

Lazy evaluation means that in order to evaluate an expression in the language, you only evaluate as much of the expression as is needed to get the final result. In contrast, strict evaluation means that all parts of an evaluation will be evaluated completely before the value of the expression as a whole is determined (with some necessary exceptions, such as if expressions, which have to be lazy to work properly).

In practice, lazy evaluation is more general, but strict evaluation is more predictable and often more efficient. Most programming languages use strict evaluation. Haskell, on the other hand, use lazy evaluation.

==

Note. One Y combinator or many?

There are infinite number of Y combinators, for now we will only be concerned with two of these, one lazy and one strict.

==

Note. Static or dynamic typing?

A statically-typed language is one where the types of all expressions are determined at combine time, and any

type errors cause the compilation to fail. A dynamically-typed language doesn't do any type checking until run time, and if a function is applied to arguments of invalid types, then an error is reported. Among commonly-used programming languages, C, C++ and Java are statically typed, and Perl, Python and Ruby and dynamically typed. Scheme is also dynamically typed.

Do not mix static/dynamic typing with weak/strong typing. Strong typing simply means that every value in the language has one and only one type, whereas weak typing means that some values can have multiple types. For example, Scheme is dynamic+strong typed; C is statically+weakly typed (because you can cast a pointer to one kind of object into a point to anther type of object without altering the pointer's value). We are only concerned with strongly typed languages here.

==

Note. What a "combinator" is

Definition. Combinator

A combinator is just a lambda expression with NO FREE VARIABLES.

A bound variable is simply a variable which is contained inside the body of a lambda expression that has the variable name as one of its arguments.

Take our factorial function as example:

```
  <rkt>
      (define factorial
        (lambda (n)
          (if (= n 0)
            1
            (* n (factorial (sub1 n))))))
  </rkt> .
```

First, we ignore the define statement, so we are left with

```
  <rkt>
      (lambda (n)
        (if (= n 0)
          1
          (* n (factorial (sub1 n)))))
  </rkt> .
```

We see that there are two variables in this lambda expression: n and factorial. Since n is a formal argument of lambda, it is bound; but factorial is not a formal argument of the lambda expression, thus we conclude that it is not a combinator.

==

Note. Abstracting out of the recursive call

Recall that our previous version of the factorial function looks like this:

```
<rkt>
     (define factorial
       (lambda (n)
          (if (= n 0)
            1
            (* n (factorial (sub1 n))))))
</rkt> .
```

What we want to do is to come up with a version of this that does the same thing but doesn't have the recursive call to factorial in the body of the function. So our first step would be saving everything else but get rid of the inner factorial function:

```
<rkt>
     (define sort-of-factorial
       (lambda (n)
          (if (= n 0)
            1
            (* n (<???> (sub1 n))))))
</rkt> .
```

When you don't know exactly what you want to put somewhere in a piece of code, just abstract it out and make it a parameter of a function. Thus we have:

```
<rkt>
     (define almost-factorial
       (lambda (f)
          (lambda (n)
            (if (= n 0)
               1
               (* n (f (sub1 n)))))))
</rkt> .
```

What we've done here is to rename the recursive call from factorial to f, and to make f an argument to a function which we're calling almost-factorial. Notice that almost-factorial is not at all factorial function. Instead, it's a higher order function which takes a single argument f, which had better be a function (otherwise (f (sub1 n)) won't make sense), and returns another function (the lambda (n) ...) part which (hopefully) will be a factorial function if we choose the right value for f.

It is important to realize that this trick can be done with any recursive function. For instance, consider a recursive function to compute fibonacci numbers. The recursive definition of fibonacci numbers is as follows:

fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci(n-1) + fibonacci(n-2)

So we can translate into Scheme like this:

```
<rkt>
  (define fibonacci
    (lambda (n)
      (cond
        [(= n 0) 0]
        [(= n 1) 1]
        [else (+ (fibonacci(sub1 n))
              (fibonacci(- n 2)))])))
</rkt> ,
```

and we can remove the explicit recursion just like we did for factorial:

```
<rkt>
  (define almost-fibonacci
    (lambda (f)
      (lambda (n)
        (cond
          [(= n 0) 0]
          [(= n 1) 1]
          [else (+ (f (- n 1))
                   (f (- n 2)))]))))
</rkt> .
```

As you can see, the transformation from a recursive function to a non-recursive almost-equivalent function is a purely mechanical one: you rename the name of the recursive function inside the body of the function to f and wrap a (lambda (f) ...) around the body.
===

Note. Sneak preview

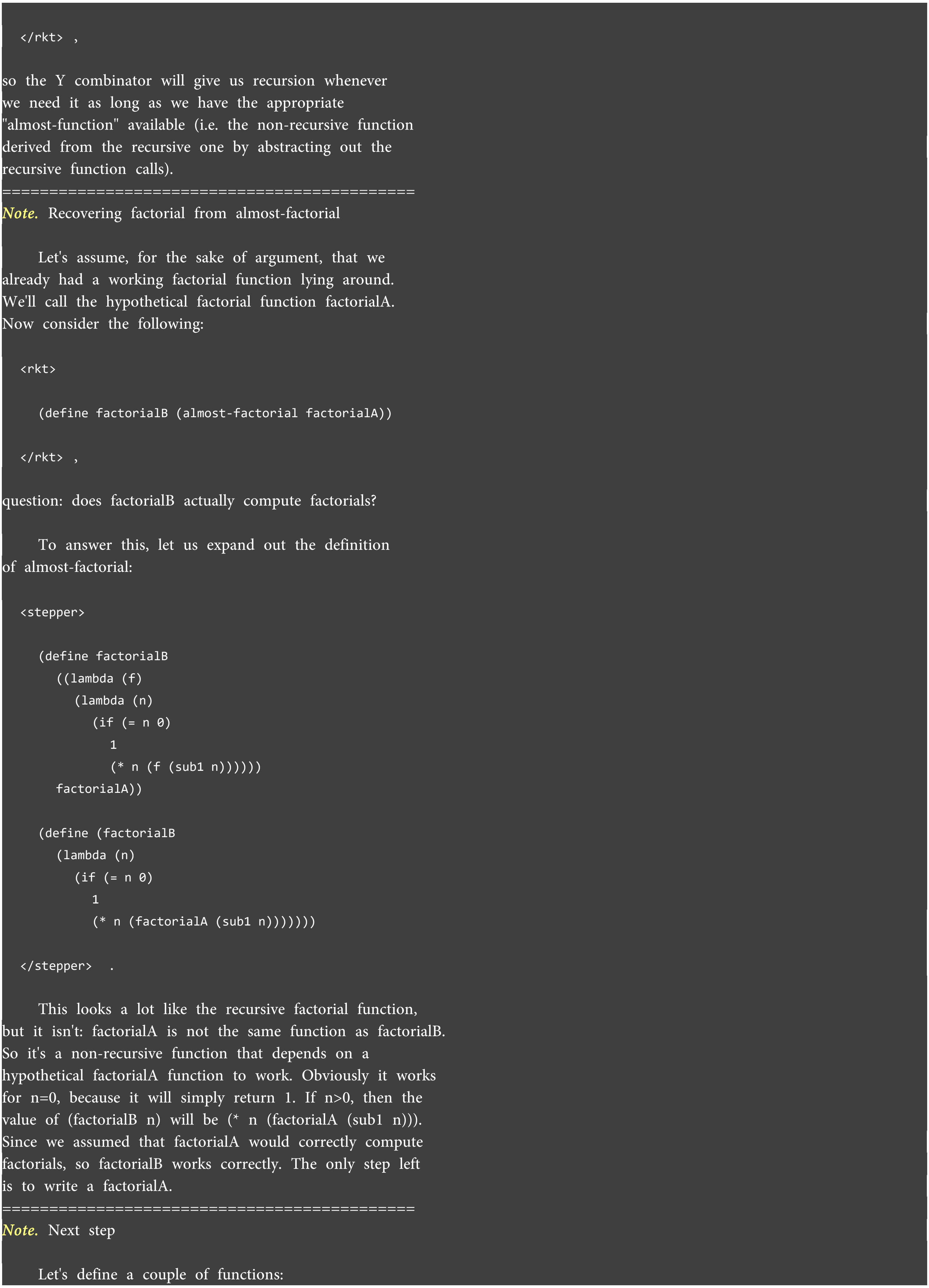
As a sneak preview of where we're going, once we have the Y combinator, we'll be able to write the factorial function using almost-factorial as follows:

```
<rkt>
  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n)))))))

  (define factorial (Y almost-factorial))

  (define almost-fibonacci
    (lambda (f)
      (lambda (n)
        (cond
          [(= n 0) 0]
          [(= n 1) 1]
          [else (+
                 (f (- n 1))
                 (f (- n 2)))]))))

  (define fibonacci (Y almost-fibonacci))
```



```
  </rkt> ,
```

so the Y combinator will give us recursion whenever we need it as long as we have the appropriate "almost-function" available (i.e. the non-recursive function derived from the recursive one by abstracting out the recursive function calls).

==

Note. Recovering factorial from almost-factorial

Let's assume, for the sake of argument, that we already had a working factorial function lying around. We'll call the hypothetical factorial function factorialA. Now consider the following:

```
  <rkt>

    (define factorialB (almost-factorial factorialA))

  </rkt> ,
```

question: does factorialB actually compute factorials?

To answer this, let us expand out the definition of almost-factorial:

```
  <stepper>

    (define factorialB
      ((lambda (f)
         (lambda (n)
           (if (= n 0)
             1
             (* n (f (sub1 n))))))
      factorialA))

    (define (factorialB
      (lambda (n)
        (if (= n 0)
          1
          (* n (factorialA (sub1 n)))))))

  </stepper>  .
```

This looks a lot like the recursive factorial function, but it isn't: factorialA is not the same function as factorialB. So it's a non-recursive function that depends on a hypothetical factorialA function to work. Obviously it works for n=0, because it will simply return 1. If n>0, then the value of (factorialB n) will be (* n (factorialA (sub1 n))). Since we assumed that factorialA would correctly compute factorials, so factorialB works correctly. The only step left is to write a factorialA.

==

Note. Next step

Let's define a couple of functions:

```
<rkt>

  (define identity (lambda (x) x))
  (define factorial0 (almost-factorial identity))

</rkt> ,
```

the identity function is pretty simply: it takes in a single argument and returns it unchanged (thus it's also a combinator!). We're basically going to use it as a placeholder when we need to pass a function as an argument and we don't know what function we should pass.

Factorial0 is more interesting. It's a function that can compute some, but not all factorials. Specifically it can compute the factorials up to and including the factorial of zero (which means that it can only compute the factorial of zero). Let's varify that:

```
<stepper>

  (factorial0 0)
=>
  ((almost-factorial identity) 0)
=>
  (((lambda (f)
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n))))))
    identity)
  0)
=>
  ((lambda (n)
    (if (= n 0)
      1
      (* n (identity (sub1 n)))))
  0)
=>
  (if (= 0 0)
    1
    (* 0 (identity (sub1 n))))
=>
  1 ;; since (= 0 0) is true.

</stepper>  ,
```

thus factorial0 can correctly compute zero's factorial. But for n>0 it will produce incorrect answers.

Now consider the next version of factorialX:

```
<rkt>

  (define factorial1
    (almost-factorial factorial0))
  </rkt> ,
```

which is the same thing as:

```
  <rkt>


    (define factorial1
      (almost-factorial
        (almost-factorial identity)))

  </rkt> ,
```

which can correctly compute the factorials of 0 and 1, but will be incorrect for any n > 1. Let's verify it again:

```
  <stepper>

    (factorial1 0)
  => ((almost-factorial factorial0) 0)
  => 1



    (factorial1 1)
  => ((almost-factorial factorial0) 1)

  => (((lambda (f)
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n))))))
      factorial0)
    1)

  => ((lambda (n)
      (if (= n 0)
        1
        (* n (factorial0 (sub1 n)))))
    1)

  =>(if (= 1 0)
      1
      (* 1 (factorial0 (sub1 1))))

  => (* 1 (factorial0 0))
  =>(* 1 1)
  => 1

  </stepper> .
```

We can keep going, and define functions which can compute factorials up to any particular limit:

```
  <rkt>
    (define factorial2 (almost-factorial factorial1))
    (define factorial3 (almost-factorial factorial2))
    ...
```

```
</rkt> .
```

One interesting way of looking at this is that almost-factorial takes in a crappy factorial function and ouputs a factorial function that is slightly less crappy, in that it will handle exactly one extra value of the input correctly. Expanding these out:

```
<rkt>

  (define factorial0 (almost-factorial identity))

  (define factorial1
    (almost-factorial
      (almost-factorial identity)))

  (define factorial2
    (almost-factorial
      (almost-factorial
        (almost-factorial identity))))

  (define factorial3
    (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial identity)))))

  (define factorial4
    (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial
            (almost-factorial identity))))))

  (define factorial5
    (almost-factorial
      (almost-factorial
        (almost-factorial
          (almost-factorial
            (almost-factorial
              (almost-factorial identity)))))))

</rkt>
```

What we've shown here is that if we would define an infinite chain of almost-factorials, that would give us the factorial function. Another way of saying this is that the factorial function if the fixpoint of almost-factorial, which is what will be explained next.

==
Note. Fixpoints of functions

Definition. Fixpoint

A fixed point of a function is an element of the function's domain that is mapped to itself by the function. That is, c is a fixed point of the function f(x) iff f(c) = c; f(f(...(f(c)...))) = $f^n(c)$ = c.

Fixed points dont'e have to be real numbers. In fact, they can be of any type of things, as long as the function that generates them can take the same type of things as input as it produces as output.

Moreover, fixpoints can be functions. If you have a higher-order function like almost-factorial that takes in a function as its input and produces a function as its ouput, then it should be possible to compute its fixpoint. That fixpoint function will be the function for which

```
<rkt>

    fix-func = (almost-factorial fix-func)

</rkt> .
```

By repeatedly substituting the right-hand side of that equation into the fix-func on the right, we get:

```
<stepper>

    fixpoint-function =
   (almost-factorial
     (almost-factorial fix-func))

   = (almost-factorial
       (almost-factorial
         (almost-factorial fix-func)))

   = ...

   = (almost-factorial
       (almost-factorial
         (almost-factorial
           (almost-factorial
             (almost-factorial ...)))))

</stepper> .
```

As we saw above, this will be the factorial function we want. Thus, the fixpoint of almost-factorial will be the factorial function:

```
<rkt>

      factorial
    = (almost-factorial factorial)
    =  (almost-factorial
        (almost-factorial
         (almost-factorial
          (almost-factorial
           ...))))

</rkt> .
```

Thus, we know that factorial is the fixpoint of almost-factorial, and we would rely on the Y combinator to tell us how to compute it. Y is also known as the fixpoint combinator: it takes in a function and returns its fixpoint.

==

Note. Y for lazy evaluation

```
  <rkt>

    ;; What Y does:
    (Y f) = fixpoint-of-f

    ;; By property of fixpoint
    (f fixpoint-of-f) = fixpoint-of-f

    ;; Combine them
    (Y f) = fixpoint-of-f = (f fixpoint-of-f)

    ;; Substitute (Y f) for fixpoint-of-f
    (Y f) = f(Y f)

    ;; Formal definition
    (define (Y f) (f (Y f)))
    (define Y (lambda (f) (f (Y f))))

  </rkt> .
```

There are two remarks to be made. First, this only works in a lazy langauge. Secondly, it is not a combinator, because the Y in the body of the definition is a free variable which is only bound once the definition is complete. In other words, we couldn't just take the body of this version of Y and plot it in whenever we need it, but it requires the name Y be defined somewhere.

Nevertheless, if you're using lazy Scheme, you can indeed define factorials like this:

```
  <rkt>

    (define Y (lambda (f) (f (Y f))))

    (define almost-factorial
      (lambda (f)
        (lambda (n)
          (if (= n 0)
            1
            (* n (f (sub1 n)))))))

    (define factorial (Y almost-factorial))

  </rkt> .
```

==

Note. Y for strict evaluation

There is a clever hack that we can use to save the day and define a version of Y that works in a strict languages. The trick is to realize that (Y f) is going to become a function of one argument. Thus, this equality will hold:

```
<rkt>

  (Y f) = (lambda (x) ((Y f) x))

</rkt> .
```

That is, no matter what argument we feed into the right side as x, it must produce the same output as the function (Y f) produces after applying it to this argument.

Now use the same approach as lazy:

```
<rkt>

  ;; What Y does
  (Y f) = fixpoint-of-f

  ;; By the property of fixpoint
  (f fixpoint-of-f) = fixpoint-of-f

  ;; substitution
  (Y f) = (f (Y f))

  ;; (Y f) = (lambda (x) ((Y f) x))
  (Y f) = (f (lambda (x) ((Y f) x)))

  ;; Thus we can do
  (define Y
    (lambda f)
      (f (lambda (x) ((Y f) x))))

</rkt>
```

Since we know that (lambda (x) ((Y f) x)) is the same function as (Y f), this is a valid version of Y which will work just as well as the previous version. Also, there is no infinite loop, because the inner (Y f) is kept inside a lambda expression, where it sits until it's needed (since the body of a lambda expression is never evaluated in Scheme until the lambda expression is applied to its arguments). Besically, you're using the lambda to delay the evaluation of (Y f). So if f was almost-factorial, we would have this:

```
<rkt>

  (define almost-factorial
    (lambda (f)
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n)))))))
```

```
    (define factorial (Y almost-factorial))

  </rkt> .
```

==

Note. Deriving the Y Combinator (LAZY)

At this point, we want to define not just Y, but a Y combinator. Note that the previous (lazy) definition of Y:

```
  <rkt>

    (define Y (lambda (f) (f (Y f))))

  </rkt>
```

is a valid definition of Y but is not a valid Y combinator, since the definition of Y refers to Y itself. In other words, this definition is explicitly recursive. A combinator isn't allowed to be explicitly recursive; it has to be a lambda expression with no free variables, which means that it can't refere to its own name in its definition.

Another way to think about this is that you should be able to replace the name of a combinator with its definition everywhere it's found and have everything still work.

Recall our original recursive factorial function:

```
  <rkt>

    (define (factorial n)
      (if (= n 0)
        1
        (* n (factorial (- n 1)))))

  </rkt> .
```

Recall that we want to define a version of this without the explicit recursion. One way we could do this is to pass the factorial function itself as an extra parameter when you call the function:

```
  <rkt>

    ;; This won't work yet:
    (define (part-factorial self n)
      (if (= n 0)
        1
        (* n (self (sub1 n)))))

  </rkt> .
```

Note that part-factorial is not the same as the almost-factorial function described above. We would have to call this part-factorial function in a different way to get it to compute factorials:

```
<rkt>

  (part-factorial part-factorial 5) ;; => 120

</rkt> .
```

This is not explicitly recursive because we send along an extra copy of the part-factorial function as the self-argument. However, it won't work unless the point of recursion calls the function the exact same way:

```
<rkt>

  (define (part-factorial self n)
    (if (= n 0)
      1
      (* n (self self (sub1 n)))))
      ;; note the extra "self" in the last line!!!

</rkt>
```

Next step,

```
<rkt>

  (define (part-factorial self)
    (lambda (n)
      (if (= n 0)
        1
        (* n ((self self (sub1 n)))))))

  ((part-factorial part-factorial) 5) ;; => 120

  (define factorial (part-factorial part-factorial))

  (factorial 5) ;; => 120

</rkt> .
```

The crucial thing here is that we defined a version of the factorial function without using explicit recursion anywhere! Every step from now on is just easy wrapping and unwrapping work.

Next, we want to get back to almost-factorial function by pulling out the (self self) call using a let expression outside of a lambda:

```
<rkt>

  (define (part-factorial self)
    (let [(f (self self))]
      (lamdba (n)
        (if (= n 0)
          1
          (* n (f (sub1 n)))))))
  </rkt> ,
```

and this is how we define the Y combinator in a lazy langauge. However in a strict language, the (self self) call in the let statement will send us into an infinite looop, because in order to calculate (part-factorial part-factorial), we need to calculate (part-factorial part-factorial) ... This doesn't cause a problem in a lazy langauge because the (self self) call in the let statement will never be evaluted unless f is actually needed (for example, when n=0 then f isn't needed to compute the answer, so (self self) won't be evaluated).

it turns out that any let expression can be converted into an equivalent lambda expression using this equation:

```
  <rkt>

    (let [(x <expr1>)]
      <expr2>)

    ;; is equivalent to

    ((lambda (x) <expr2>) <expr1>)

  </rkt> .
```

This leads us to:

```
  <rkt>

    (define (part-factorial self)
      ((lambda (f)
         (lambda (n)
           (if (= n 0)
             1
             (* n (f (sub1 n))))))
      (self self)))

  </rkt> .
```

Look closely, you will find our old friend almost-factorial is embedded inside the part-factorial function.

```
  <rkt>

    (define almost-factorial
      (lambda (f)
        (lambda (n)
          (if (= n 0)
            1
            (* n (f (sub1 n)))))))

    (define (part-factorial self)
      (almost-factorial (self self)))

    (define factorial (part-factorial part-factorial))
  </rkt> .
```

The "part-factorial" is really messy, so let's rewrite it:

```
  <rkt>

    (define part-factorial
      (lambda (self)
        (almost-factorial
          (self self))))

  </rkt> .
```

Then I can rewrite the factorial function like this:

```
  <rkt>

    (define almost-factorial
      (lambda (f)
        (lambda (n)
          (if (= n 0)
            1
            (* n (f (sub1 n)))))))

    (define factorial
      (let
        [(part-factorial
          (lambda (self)
            (almost-factorial (self self))))]
        (part-factorial part-factorial)))

    ;; change part-factorial to x
    (define factorial
      (let
        [(x (lambda (self)
            (almost-factorial (self self))))]
        (x x)))

    ;; use the same let => lambda trick we used before
    (define factorial
      ((lambda (x) (x x))
        (lambda (self)
          (almost-factorial (self self)))))

    ;; rename self to x
    (define factorial
      ((lambda (x) (x x))
      (lambda (x) (almost-factorial (x x)))))

    ;; finally we can obtain our Y combinator
    (define Y
      ((lambda (x) (x x))
       (lambda (x) (f (x x)))))

    ;; use Y to rewrite everything
    (define (Y f)
      ((lambda (x) (x x))
       (lambda (x) (f (x x)))))
    (define almost-factorial
      (lambda (f)
        (lambda (n)
          (if (= n 0)
            1
            (* n (f (sub1 n)))))))

    ;; expand out the definition of Y a bit:
    (define Y
      (lambda (f)
        ((lambda (x) (x x))
         (lambda (x) (f (x x))))))

    ;; apply the inner lambda expression to its
    ;; argument to get an equivalent version of Y:
    (define Y
      (lambda (f)
        ((lambda (x) (f (x x)))  ;; think about this.
         (lambda (x) (f (x x))))))

  </rkt>
```

What this means is that, for a given function f (which is a non-recursive function like almost-facotiral), the corresponding recursive function can be obtained first by combputing (lambda (x) (f (x x))), and then applying this lambda expression to itself. This is the usual definition of the normal-order Y combinator.

The last step is to demonstrate that this equation is correct:

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

```
  <rkt>

    ;; from the definition of the normal-order Y
      (Y f)
    = ((lambda (x) (f (x x)))
       (lambda (x) (f (x x))))

    ;; now apply the first lambda expression to its argument,
    ;; which is the second lambda expression, we have:
    = (f ((lambda (x) (f (x x)))
          (lambda (x) (f (x x))))
    = (f (Y f))

  </rkt> .
```

==

Note. Deriving the Y Combinator (STRICT)

Let's pick up the previous derivation just before the point where it failed for strict languages:

```
  <rkt>
```

```
    (define (part-factorial self)
      (lambda (n)
        (if (= n 0)
          1
          (* n ((self self) (sub1 n))))))

    (define factorial (part-factorial part-factorial))

  </rkt>
```

Now if we pull "self self" out into a let statement, it won't work because we are using strict evaluation. To fix this, we wrap everything up with a lambda.

```
  <rkt>

    (define (part-factorial self)
      (let [(f (lambda (y) (self self) y))]
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n))))))

    (define factorial (part-factorial part-factorial))

  </rkt> .
```

From this we can derive the definition of the strict Y combinator:

```
  <rkt>

    (define Y
      (lambda (f)
        ((lambda (x) (x x))
         (lambda (x) (f (lambda (y) ((x x) y)))))))

    ;; or

    (define Y
      (lambda (f)
        (lambda (x) (f (lambda (y) ((x x) y))))
        (lambda (x) (f (lambda (y) ((x x) y))))))

  </rkt> .
```

Back to the original problem, using the newly-defined Y combinator, we can define our factorial function like this:

```
  <rkt>

    (define almost-factorial
      (lambda (f)
        (lambda (n)
          (if (= n 0)
            1
            (* n (f (sub1 n)))))))
    (define factorial (Y almost-factorial))


  </rkt> .


==============================================
==============================================
END
```