Topic. Y Combinator II

Implementation. Strict Y-combinator

```
<rkt>
  (define (Y f)
    ((λ(x) (f (x x)))
     (λ(x) (f (x x)))))
</rkt>
```

Note. Why it works.

What Y does: generate the fixpoint of function f.
Thus (Y f) should give us a magical value c, where f(c) = c.
Replace c with (Y f) we get (f (Y f)) = (Y f).

Remark.

```
<rkt>
  ;; Let g = (Y f) be the fixpoint of function f.
  ;; Note that the functions below are equivalent:
  g = (lambda (x) (g x))

  ;; Replace g with (Y f), we get
  (Y f) = (lambda (x) ((Y f) x))

  ;; And since (Y f) = (f (Y f)), we get
  (Y f) = (f (lambda (x) ((Y f) x)))

  ;; From this we derive the beta version of Y:
  (define (Y f)
    (f (lambda (x) ((Y f) x))))

  ;; Abstract out parameter f:
  (define Y
    (lambda (f)
      (f (lambda (x) ((Y f) x)))))
</rkt>
==========================================================================
```

Note. Derivation

1. Normal factorial function:

```
  <rkt>
    (define (factorial n)
      (if (zero? n)
        1
        (* n (factorial (sub1 n)))))
  </rkt>
```

2. Rewrite using all lambdas:

```
  <rkt>
    (λ (factorial)
      (λ (n)
      (if (zero? n)
         1
         (* n (factorial (sub1 n))))))
</rkt>
```

3. Note that we need to pass a factorial function as parameter into this lambda expression to make it work. But this lambda expression was supposed to be the factorial function, so why don't we pass itself as the parameter?

```
<rkt>
  ((λ (factorial)
    (λ (n)
      (if (zero? n)
         1
         (* n (factorial (sub1 n))))))
   (λ (factorial)
    (λ (n)
      (if (zero? n)
         1
         (* n (factorial (sub1 n)))))))
</rkt>
```

4. Since we are literally applying the function to itself, we can rewrite it like this:

```
<rkt>
  ((λ (f)          ;; we take in a function
    (f f))         ;; and apply the function to itself
   (λ (factorial)  ;; below are the factorial function
    (λ (n)
      (if (zero? n) 1
         (* n (factorial (sub1 n)))))))
</rkt>
```

5. Don't forget what we have here is still just an anonymous function, we need an extra argument "n" to compute an answer.

6. Let's write a stepper and see if it's correct. Let n = 5 and evaluate the function.

```
<rkt>
  (((λ (f)
     (f f))
   (λ (factorial)
    (λ (n)
      (if (zero? n) 1
         (* n (factorial (sub1 n))))))) 5)
</rkt>

<stepper>
;; substitution: factorial => f
=> ((λ (n)
    (if (zero? n) 1
      (* n
         ((λ (factorial)
                (λ (n)
                  (if (zero? n) 1
                    (* n (factorial (sub1 n))))))
                (sub1 n))))) 5)

;; substitution: 5 => n
=> (if (zero? 5) 1
     (* 5 ((λ (factorial)
             (λ (n)
               (if (zero? n) 1
                 (* n (factorial (sub1 n))))) (sub1 5))))

;; if-statement evaluates to false
=> (* 5 ((λ (factorial)
           (λ (n)
             (if (zero? n) 1
               (* n (factorial (sub1 n)))))) 4))

;; substitution: 4 => factorial
;; Note that this step is obviously wrong. 4 is not a function.
;; From this we can conclude that we are trying to apply (sub1 n),
;; which, in this case, is (sub1 5), as the factorial function.
;; What we were supposed to do is to first pass in the function
;; "factorial", then pass in the value (sub1 n).
</stepper>
```

7. Consider the following functions:

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

  ;; THIS DOES NOT WORK
  ;; SAME REASON AS STEP 6
  (define (partial-factorial self n)
    (if (= n 0) 1
      (* n (self (sub1 n)))))

  (partial-factorial partial-factorial 3)
</rkt>

<stepper>
~> (partial-factorial partial-factorial 3)

=> (if (= 3 0) 1
     (* 3 (partial-factorial <??> 2)))

=> (* 3
     (if (= 2 0) 1
       (* 2 (partial-factorial <??> 1))))

=> (* 3
     (* 2
       (if (= 1 0) 1
          (* 1 (partial-factorial <??> 0)))))

=> (* 3
     (* 2
       (* 1
         (if (= 0 0) 1
           ...))))

=> (* 3 (* 2 (* 1 1)))

=> 6
</stepper>
```

8. In the function above, we passed in the function partial-factorial itself as one of the argument to the function application. Note that partial-factorial takes in 2 parameters, the first one is a copy of itself, the second one is the number n. Thus to make the recursive line work (the else-clause in each if-statement), we need to pass in another copy of partial-factorial at all the places of <??>. Now we can finally write a correct version of partial-factorial.

```
<rkt>
  (define (partial-factorial self)
    (lambda (n)
      (if (= n 0) 1
        (* n ((self self) (sub 1 n))))));; note the extra "self" on this line!!

;; Remark:
;; In the previous application, (partial-factorial self n) takes in 2 parameters,
;; thus we need to do (partial-factorial partial-factorial 3). But in this implementation,
;; we pulled out n from the function definition, thus (partial-factorial self) consumes
;; only one parameter, so (partial-factorial partial-factorial) is what we needed.
;; Applying this to 3, we need to do ((partial-factorial partial-factorial) 3).
</rkt>
```

9. Use local definition then lambda to simplify this function.

```
<rkt>
  (define (partial-factorial self)
    (lambda (n)
      (if (= n 0) 1
        (* n ((self self) (sub 1 n))))))

  (define (partial-factorial self)
    (let [(f (self self))] ;; we replace (self self) with f
      (lambda (n)
        (if (= n 0) 1
          (* n (f (sub1 n)))))))

  (define (partial-factorial self)
    ((lambda (f)
      (lambda (n)
        (if (= n 0)
          1
```

```
            (* n (f (sub1 n))))))
      (self self))) ;; we pass in (self self) as the first parameter, aka f.
</rkt>
```

10. Since we know ((partial-factorial partial-factorial) n) computes the correct answer, We can now define the factorial function.

```
<rkt>
  (define factorial (partial-factorial partial-factorial))
</rkt>
```

11. Look closely at the last definition in step 9, you will recognize that our old friend (which we defined in step 2) appears inside the definition.

```
<rkt>
  ;; Give step 2 a name:
  (define almost-factorial
    (λ (factorial)
      (λ (n)
        (if (zero? n)
          1
          (* n (factorial (sub1 n)))))))

  ;; What we had in step 9
  (define (partial-factorial self)
    ((lambda (f)
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n))))))
     (self self)))

  ;; Use almost-factorial to rewrite partial-factorial
  (define (partial-factorial self)
    (almost-factorial (self self)))

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

  ;; Now define factorial:
  (define factorial (partial-factorial partial-factorial))

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

  ;; rewrite function using let
  (define factorial
    (let [(x (lambda (self) (almost-factorial (self self))))]
      (x x)))
  ;; rewrite let using lambda
  (define factorial
    ((lambda (x) (x x))
      (lambda (self) (almost-factorial (self self)))))

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

  ;; finally we have our y
  (define Y
    (lambda (f)
      ((lambda (x) (x x)
       (lambda (y) (f (y y)))))))

  ;; pass in (lambda (y) (f (y y))) as x to (lambda (x) (x x))
  (define Y
    (lambda (f)
      ((lambda (y) (f (y y)))
       (lamdba (y) (f (y y))))))
</rkt>
```

12. Prove that this definition is correct.

```
<rkt>
  (Y f) = ((lambda (y) (f (y y)))
           (lambda (y) (f (y y))))

  ;; Now pass in second line (lambda (y) (f (y y))) to the first line
  ;; (lambda (y) (f (y y)))
        = (f (lambda (y) (f (y y)))
             (lambda (y) (f (y y))))
        = (f (Y f))

  ;; Hence our Y is correct.
</rkt>
```

13. However the above Y only works for lazy evaluation, since (self self) would result in an infinite loop in strict order of evaluation. We use the fact that g = (lambda (x) (g x)) to help us solve the problem.

```
<rkt>
  ;; From step 11
  (define (partial-factorial self)
    ((lambda (f)
      (lambda (n)
        (if (= n 0)
          1
          (* n (f (sub1 n))))))
     (self self))) ;; this line is causing us problems.

  ;; Since f = (lambda (x) (f x)), we can rewrite it as this:
  (define (partial-factorial self)
      ((lambda (f)
         (lambda (n)
           (if (= n 0)
               1
               (* n (f (sub1 n))))))
       (lambda (k) ((self self) k))))

    ;; So factorial would look like this:
    (define factorial
      ((lambda (x) (x x))
         (lambda (y) (almost-factorial (lambda (k) ((y y) k))))))


    ;; And we modify our Y combinator accordingly:
    (define Y
      (lambda (f)
        ((lamdba (x) (x x)
          (lambda (y) (f (lambda (k) ((y y) k))))))))

    (define Y
      (lambda (f)
        ((lambda (y) (f (lambda (k) ((y y) k))))
          (lambda (y) (f (lambda (k) ((y y) k)))))))

  </rkt>
```

14. And this is the final version of Y-combinator.
==
END