```
;;;;        Y operators, by GJS

;;; We are looking for a Y that computes a
;;;  fixed-point of F, i.e.  a Y such that
;;;  (Y F) = (F (Y F))

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

;;; This won't work, but it has the idea!
;;; We remove the self-reference by making a
;;;  clever infinite loop.  Here we have
;;;  (Y F) = (F (F (F (F ...)))) = (F (Y F))


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

;;; Unfortunately, the above does not work in
;;;  applicative order -- it converges in
;;;  normal order.
```

```
;;; The following Y operator finds a least
;;;  fixed point of a kernel in applicative
;;;  order.  We control the order of
;;;  evaluation by an eta abstraction

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

(define fact-kernel
  (lambda (fact)
    (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1)))))))

;;; e.g.

(define fact (Y fact-kernel))

(fact 4)
;Value: 24
```

```
;;; Alternatively,

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


;;; e.g.

(define fact (Y fact-kernel))

(fact 4)
;Value: 24
```

```
;;; but this does not work for multi-argument
;;;  procedures

(define expt-kernel
  (lambda (expt)
    (lambda (x n)
      (if (= n 0)
          1
          (* x (expt x (- n 1)))))))


;;; We can fix this problem with a simple
;;;  syntactic device.

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

(define exptf (Y expt-kernel))

;;;this does work.

(exptf 2 3)
;Value: 8
```

```
;;; A cheap way of doing recursion...

(let ((f1
       (lambda (f n)
         (if (< n 2)
             1
             (* n (f f (- n 1)))))))
  (f1 f1 5))
;Value: 120

(let ((f1
       (lambda (f n)
         (if (< n 2)
             n
             (+ (f f (- n 2))
                (f f (- n 1)))))))
  (f1 f1 20))
;Value: 6765
```

```
;;; Pulling the kernel out

(let ((f1
       (lambda (f)
         (lambda (n)
           (if (< n 2)
               1
               (* n ((f f) (- n 1))))))))
  (let ((fact (f1 f1)))
    (fact 5)))
;Value: 120


;;; Letrec for corecursion...

(let ((f1
       (lambda (f1 f2)
         (lambda (n)
           (if (< n 2)
               1
               (* n ((f1 f1 f2) (- n 1)))))))
      (f2
       (lambda (f1 f2)
         (lambda (n)
           (if (< n 2)
               n
               (+ ((f2 f1 f2) (- n 1))
                  ((f2 f1 f2) (- n 2))))))))
  (let ((fact (f1 f1 f2))
        (fib (f2 f1 f2)))
    (+ (fact 5) (fib 10))))
;Value: 175
```