We are OUDL.

Organization for the Understanding of Dynamic Languages

http://meetup.com/dynamic/

We are programming language enthusiasts
Check us out on Meetup.com
All events have been by members, for members
Each event has a theme, a selected pastry or baked good, and a horrible logo

What makes Objective C dynamic?

What makes Objective C dynamic?

Kamehameha Bakery donuts

Otto Cake cheesecake

Cake Couture cupcakes

Fendu Bakery croissants & cookies

Saint Germain Bakery palmiers

Mahalo.

HTDC
Cultivating
Hawaii's tech sector

THE BOX JELLY

HICAP
hicapacity.org

Y combinator

Examples in Clojure.
Also includes: blenders and kittens.
Caveat emptor: I make no effort to teach you Clojure.

Kyle Oba
@mudphone
Pas de Chocolat

Not this one.

Paul Graham did name his company after the *REAL* Y combinator.
But, why?

This one.

```
(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))
```


Which is this thing, in Clojure.

Let's get started.

Here's a non-recursive definition of factorial, using the Y combinator.

Here it is. Thank you, good night and good luck.

```clojure
(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))

(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))

(defn factorial
  [n]
  ((Y almost-factorial) n))
```


Here it is. Thank you, good night and good luck.

Questions?

An example: 5!

```
(factorial 5) = 5 * 4 * 3 * 2 * 1 = 120
```

```
(defn factorial
  [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```


Let's back up. This is how a sane person would define factorial... recursively.

here to here?

```
(defn factorial
  [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

```
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

```
(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))

(defn factorial
  [n]
  ((Y almost-factorial) n))
```


Recursive definition to non-recursive

2 Things

1) recursion

2) functions

2 Things

1) recursion

2) functions

"The Y combinator allows recursion...
as a set of ***rewrite rules***,

without requiring native recursion
support in the language."

-- Someone on Wikipedia

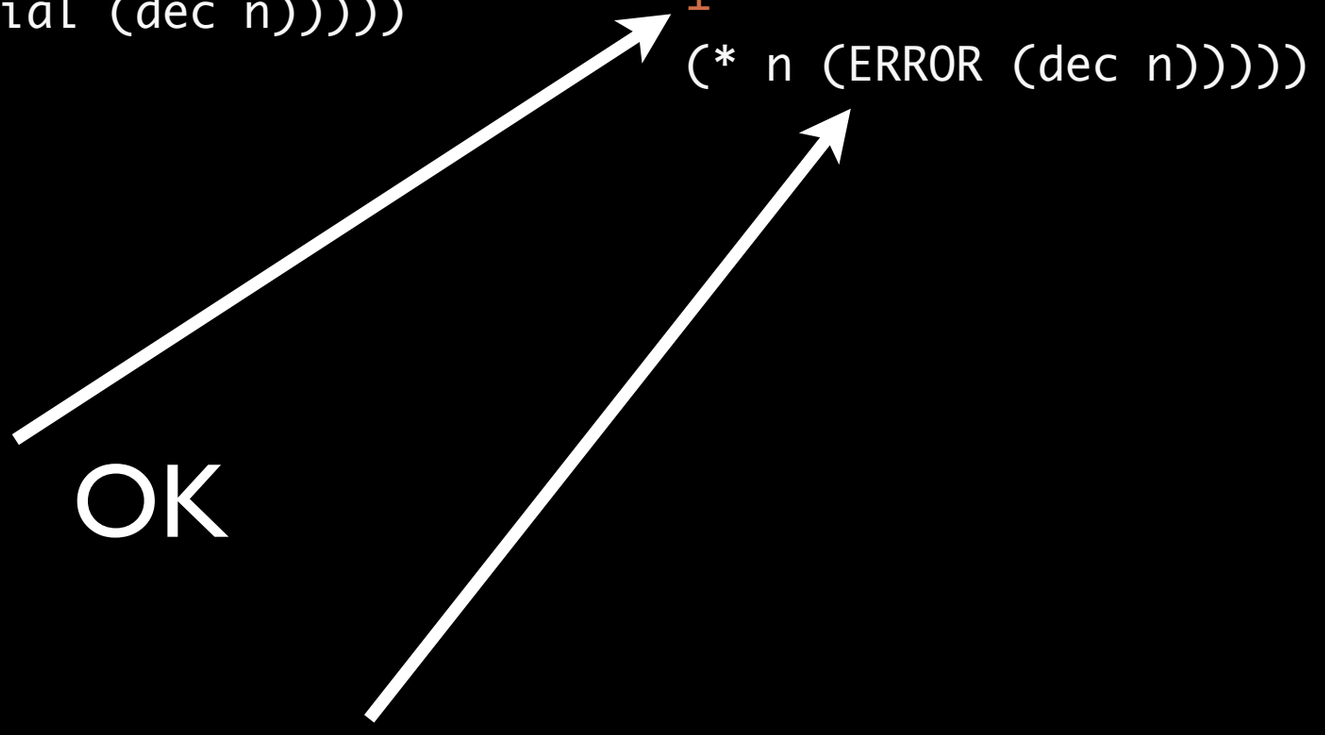
replace
“native recursion”
with
manual recursion

```clojure
(defn factorial
  [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

```clojure
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

```clojure
(defn factorial
  [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

```clojure
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```



n = 0 OK

n = 1 BOOM!

```
(defn factorial
  [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

n = 0 OK

n = 1 BOOM!

```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)
```

```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

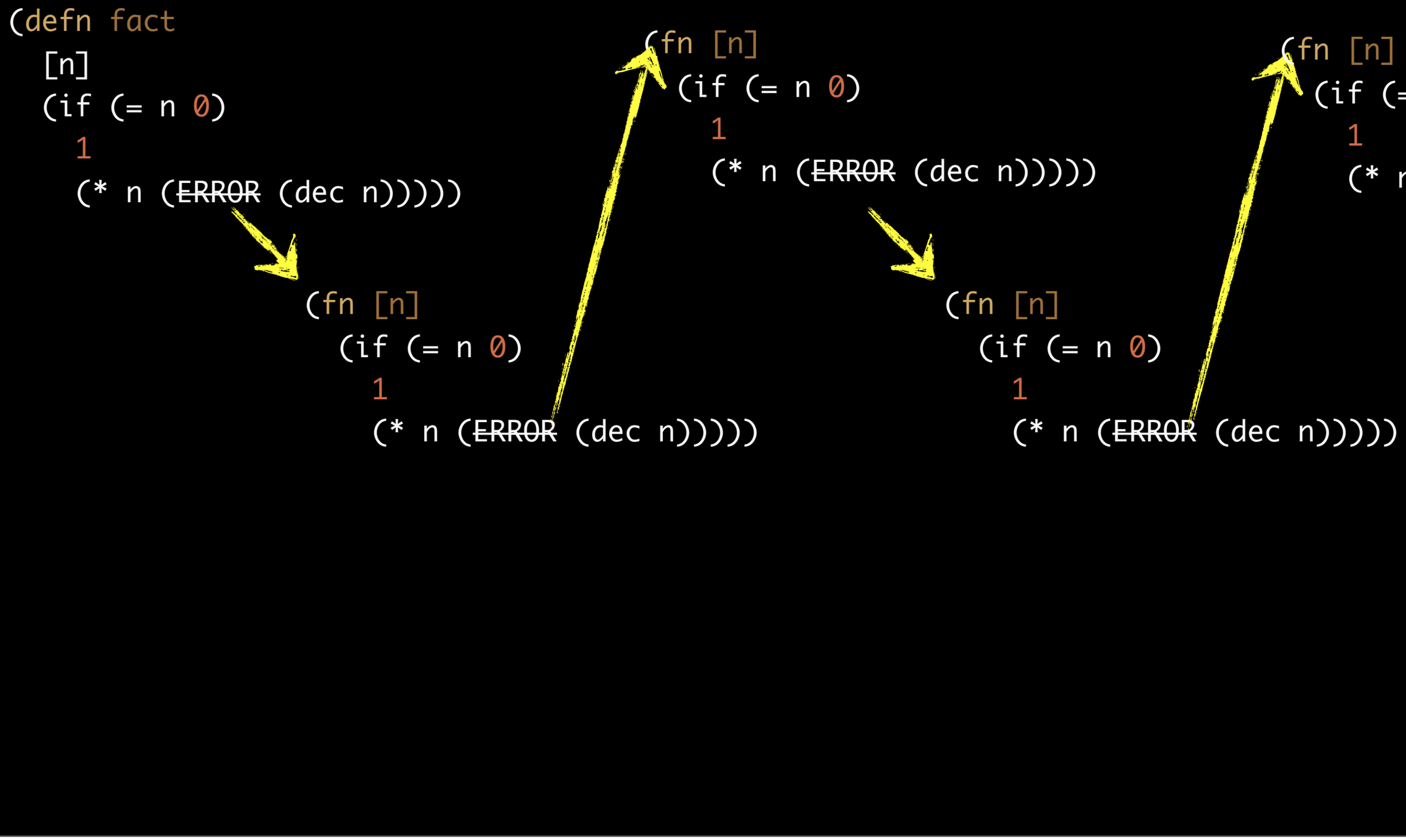
```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)
```

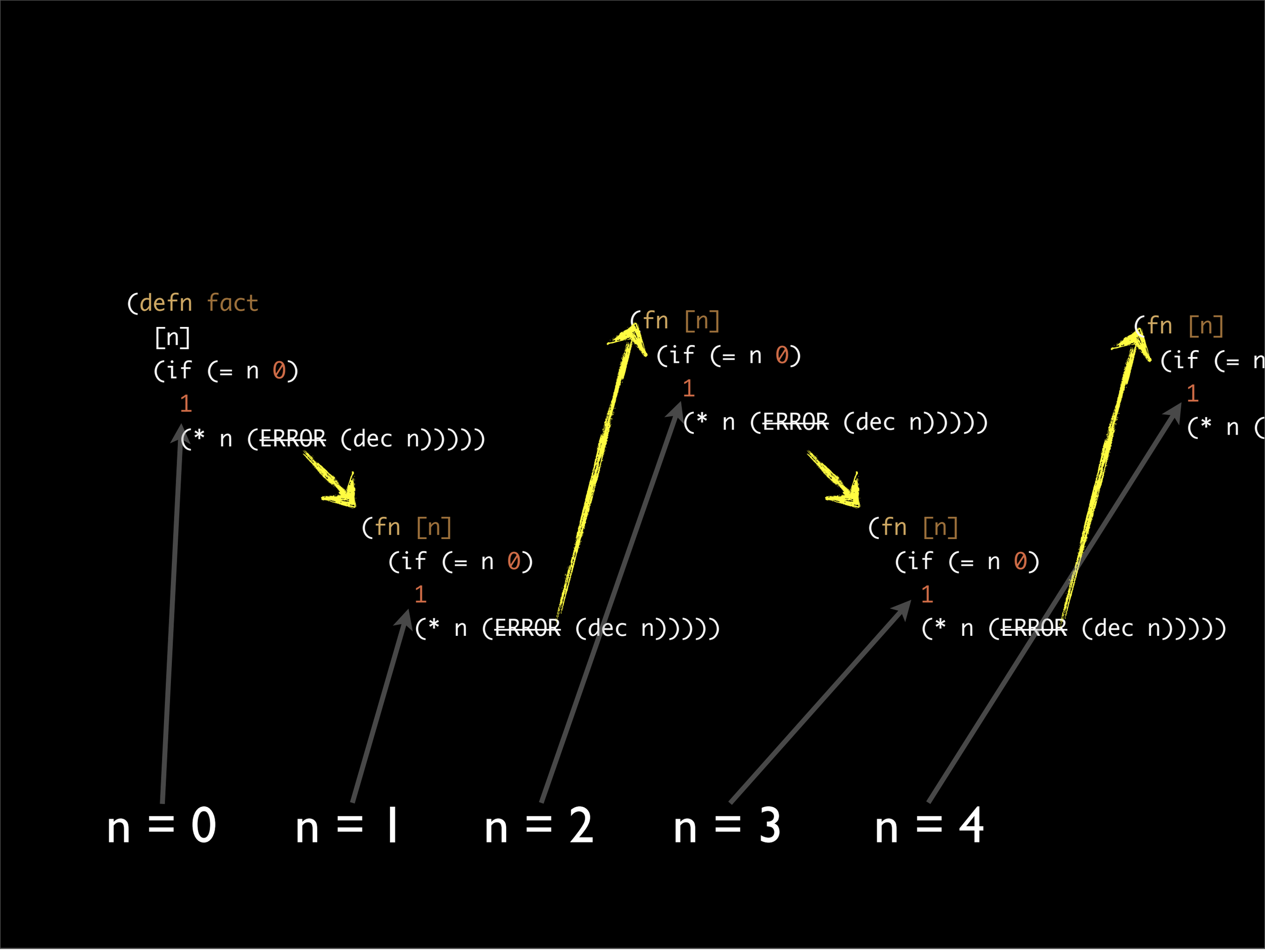
```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)
```

```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

```
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)
```


(defn fact
[n]
(if (= n 0)
1
(* n (ERROR (dec n)))))
(fn [n]
(if (= n 0)
1
(* n (ERROR (dec n)))))
(fn [n]
(if (= n 0)
1
(* n (ERROR (dec n)))))
(fn [n]
(if (= n
1
(* n (
(if (= n 0)
1
(* n (ERROR (dec n)))))
(if (= n 0)
1
(* n (ERROR (dec n)))))


(defn fact
[n]
(if (= n 0)
1
(* n (ERROR (dec n)))))
(fn [n]
(if (= n 0)
1
(* n (ERROR (dec n)))))
(fn [n]
(if (= n 0)
1
(* n (ERROR (dec n)))))
(fn [n]
(if (= n
1
(* n (
n = 0
n = 1
n = 2
n = 3
n = 4

replace
“native recursion”
with
~~manual recursion~~
“rewrite rules”

2 Things

1) recursion

2) functions

2 Things

1) recursion

2) functions

Functions are machines.

Functions are relationships,
between inputs and outputs.

A function is a blender.

FIRST ORDER BLENDER

A normal blender that consumes single input and creates output.

FIRST ORDER BLENDER

A normal blender that consumes single input
and creates output.

HIGHER ORDER BLENDER

A special blender that consumes a blender
and outputs another blender.

FIRST ORDER BLENDER

A normal blender that consumes single input
and creates output.

HIGHER ORDER BLENDER

A special blender that consumes a blender
and outputs another blender.

FIXPOINT (BLENDER) COMBINATOR

Consumes a blender and produces a new blender that
can consume any number of inputs.

ONE

ONE

```
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

ONE

```
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

ANY

ONE

```
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

Y

ANY

factorial

if you squint

```clojure
(defn factorial
  [n]
  (if (= n 0)
    1
    (* n (factorial (dec n)))))
```

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

```clojure
(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))

(defn factorial
  [n]
  ((Y almost-factorial) n))
```


Derivation not possible in 3 minutes.

I'm so sorry.

No kittens were blended during the creation of this presentation.

No really, done now.

No kittens were blended during the creation of this presentation.

A Clojure Primer

PARENTHESIS

```
(+ 1 2 3)
;; => 6
```

PREFIX NOTATION

```
(operator arg1 arg2 arg3)
```

FUNCTIONS

```
(defn multby2
  [n]
  (* n 2))
;; (multby2 4) => 8
```

```
(fn [n] (* n 2))
```

Wednesday, August 22, 12

1) First, a primer on LISP & Clojure
- parens for function call
- prefix notation, followed by arguments
2) And, function definition and anonymous functions

```clojure
(defn simple-factorial
  [n]
  (if (= n 0)
    1
    (* n (simple-factorial (dec n)))))
```


Here, we remove the recursive definition. Kind of delaying it, for now.

```clojure
(defn simple-factorial
  [n]
  (if (= n 0)
    1
    (* n (simple-factorial (dec n)))))


(defn part
  [self n]
  (if (= n 0)
    1
    (* n (self self (dec n)))))
;; (part part 5) => 120
```


Here, we remove the recursive definition. Kind of delaying it, for now.

```clojure
(defn simple-factorial
  [n]
  (if (= n 0)
    1
    (* n (simple-factorial (dec n)))))


(defn part
  [self n]
  (if (= n 0)
    1
    (* n (self self (dec n)))))
;; (part part 5) => 120
```


Change part to take a single arg, returning a function that takes n.

```clojure
(defn part
  [self n]
  (if (= n 0)
    1
    (* n (self self (dec n)))))
;; (part part 5) => 120

(defn part2
  [self]
  (fn [n]
    (if (= n 0)
      1
      (* n ((self self) (dec n))))))
;; ((part2 part2) 5) => 120
```


Change part to take a single arg, returning a function that takes n.

```clojure
(defn part
  [self n]
  (if (= n 0)
    1
    (* n (self self (dec n)))))
;; (part part 5) => 120

(defn part2
  [self]
  (fn [n]
    (if (= n 0)
      1
      (* n ((self self) (dec n))))))
;; ((part2 part2) 5) => 120
```


Replace (self self) with f, which blows up in a Stack Overflow... but, we press on.

```clojure
(defn part2
  [self]
  (fn [n]
    (if (= n 0)
      1
      (* n ((self self) (dec n))))))
;; ((part2 part2) 5) => 120

(defn part3
  [self]
  (let [f (self self)]
    (fn [n]
      (if (= n 0)
        1
        (* n (f (dec n)))))))
```


Replace (self self) with f, which blows up in a Stack Overflow... but, we press on.

```clojure
(defn part2
  [self]
  (fn [n]
    (if (= n 0)
      1
      (* n ((self self) (dec n))))))
;; ((part2 part2) 5) => 120

(defn part3
  [self]
  (let [f (self self)]
    (fn [n]
      (if (= n 0)
        1
        (* n (f (dec n)))))))
```


Bury, the (self self) call in a lambda.

```clojure
(defn part3
  [self]
  (let [f (self self)]
    (fn [n]
      (if (= n 0)
        1
        (* n (f (dec n)))))))

(defn part4
  [self]
  (let [f (fn [y] ((self self) y))]
    (fn [n]
      (if (= n 0)
        1
        (* n (f (dec n)))))))
```


Bury, the (self self) call in a lambda.

```clojure
(defn part3
  [self]
  (let [f (self self)]
    (fn [n]
      (if (= n 0)
        1
        (* n (f (dec n)))))))

(defn part4
  [self]
  (let [f (fn [y] ((self self) y))]
    (fn [n]
      (if (= n 0)
        1
        (* n (f (dec n)))))))
```

Wednesday, August 22, 12

Rip out the function that looks almost like the factorial function. This is what we're generalizing. The Y combinator computes the fixpoint of this function.

```clojure
(defn part4
  [self]
  (let [f (fn [y] ((self self) y))]
    (fn [n]
     (if (= n 0)
       1
       (* n (f (dec n)))))))))

(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```


Rip out the function that looks almost like the factorial function. This is what we're generalizing. The Y combinator computes the fixpoint of this function.

```clojure
(defn part4
  [self]
  (let [f (fn [y] ((self self) y))]
    (fn [n]
     (if (= n 0)
       1
       (* n (f (dec n)))))))))

(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```


Insert almost-factorial into the part function.

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))


(defn part5
  [self]
  (let [f (fn [y] ((self self) y))]
    (almost-factorial f)))
```


Insert almost-factorial into the part function.

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))


(defn part5
  [self]
  (let [f (fn [y] ((self self) y))]
    (almost-factorial f)))
```


fact5 is a working factorial function, but we can generalize it

```clojure
(defn part5
  [self]
  (let [f (fn [y] ((self self) y))]
    (almost-factorial f)))



(defn fact5
  [n]
  ((part5 part5) n))
```


fact5 is a working factorial function, but we can generalize it

```clojure
(defn part5
  [self]
  (let [f (fn [y] ((self self) y))]
    (almost-factorial f)))



(defn fact5
  [n]
  ((part5 part5) n))
```


here we embed the definition the “part” function

```clojure
(defn fact5
  [n]
  ((part5 part5) n))
```

```clojure
(def fact6
  (let [part (fn [self]
               (let [f (fn [y] ((self self) y))]
                 (almost-factorial f)))]
    (part part)))
```


here we embed the definition the “part” function

```clojure
(defn fact5
  [n]
  ((part5 part5) n))
```

```clojure
(def fact6
  (let [part (fn [self]
               (let [f (fn [y] ((self self) y))]
                 (almost-factorial f)))]
    (part part)))
```


rename part => x
and self => x
for kicks really

```clojure
(def fact6
  (let [part (fn [self]
               (let [f (fn [y] ((self self) y))]
                 (almost-factorial f)))]
    (part part)))



(def fact7
  (let [x (fn [x]
            (let [f (fn [y] ((x x) y))]
              (almost-factorial f)))]
    (x x)))
```


rename part => x
and self => x
for kicks really

```
(def fact6
  (let [part (fn [self]
               (let [f (fn [y] ((self self) y))]
                 (almost-factorial f)))]
    (part part)))



(def fact7
  (let [x (fn [x]
            (let [f (fn [y] ((x x) y))]
              (almost-factorial f)))]
    (x x)))
```


replace the (x x) invocation with a lambda of the same

```clojure
(def fact7
  (let [x (fn [x]
            (let [f (fn [y] ((x x) y))]
              (almost-factorial f)))]
    (x x)))



(def fact8
  ((fn [x]
     (x x)) (fn [x]
              (let [f (fn [y] ((x x) y))]
                (almost-factorial f)))))
```


replace the (x x) invocation with a lambda of the same

```clojure
(def fact7
  (let [x (fn [x]
            (let [f (fn [y] ((x x) y))]
              (almost-factorial f)))]
    (x x)))



(def fact8
  ((fn [x]
     (x x)) (fn [x]
              (let [f (fn [y] ((x x) y))]
                (almost-factorial f)))))
```


Rename to Y

and generalize, by accepting a function g and using this to replace almost-factorial

```clojure
(def fact8
  ((fn [x]
     (x x)) (fn [x]
              (let [f (fn [y] ((x x) y))]
                (almost-factorial f)))))



(defn nearly-Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (let [f (fn [y] ((x x) y))]
                      (g f)))))
```


Rename to Y
and generalize, by accepting a function g and using this to replace almost-factorial

```clojure
(def fact8
  ((fn [x]
     (x x)) (fn [x]
              (let [f (fn [y] ((x x) y))]
                (almost-factorial f)))))



(defn nearly-Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (let [f (fn [y] ((x x) y))]
                      (g f)))))
```


Replace f with the anonymous function bound to it

```clojure
(defn nearly-Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (let [f (fn [y] ((x x) y))]
                      (g f)))))



(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))
```


Replace f with the anonymous function bound to it

```
(defn nearly-Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (let [f (fn [y] ((x x) y))]
                      (g f)))))



(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))
```

```clojure
(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))


(defn factorial
  [n]
  ((Y almost-factorial) n))
```

```
(defn Y
  [g]
  ((fn [x] (x x)) (fn [x]
                    (g (fn [y] ((x x) y))))))
```

I'm so sorry.

```
(defn factorial
  [n]
  ((Y almost-factorial) n))
```

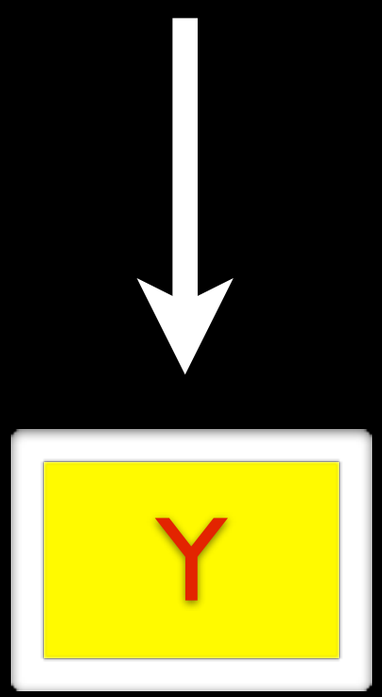
I'm so sorry.

No kittens were blended during the creation of this presentation.

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

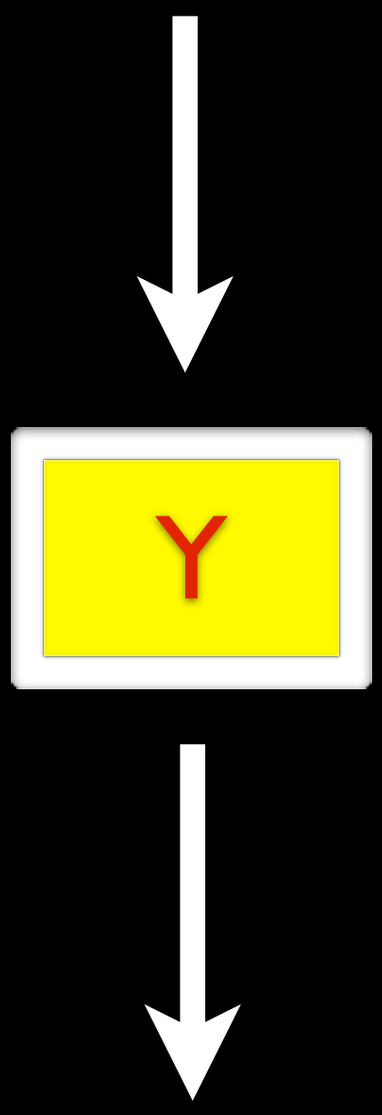
```clojure
(Y almost-factorial)
;; ((Y almost-factorial) 5) => 120
```

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```




```clojure
(Y almost-factorial)
;; ((Y almost-factorial) 5) => 120
```

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

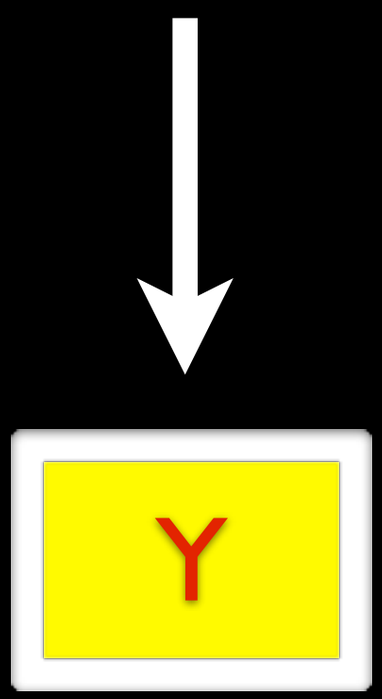



```clojure
(Y almost-factorial)
;; ((Y almost-factorial) 5) => 120
```

FACTORIAL

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

```clojure
(Y almost-factorial)
;; ((Y almost-factorial) 5) => 120
```




```clojure
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

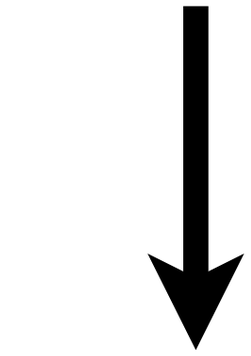
```clojure
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

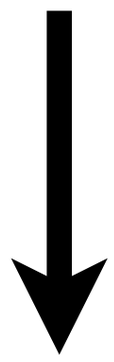
```clojure
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

```clojure
(defn fact
  [n]
  (if (= n 0)
    1
    (* n (ERROR (dec n)))))
```

FACTORIAL

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```





factorial

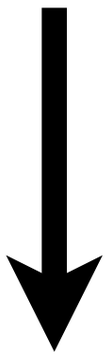
```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

Y

factorial

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

ONE



factorial

```clojure
(defn almost-factorial
  [f]
  (fn [n]
    (if (= n 0)
      1
      (* n (f (dec n))))))
```

Y

factorial

ONE

ANY