What solves this equation?

Equation:

$$\texttt{fact} = \lambda n.\text{if } n = 0 \text{ then } 1 \text{ else } n \times \texttt{fact}(n-1)?$$

The factorial function!

Factorial in lambda calculus

Wish for:

```
fact = \n.(zero? n) 1 (times n (fact (pred n)));
```

But: on right-hand side, `fact` is not defined.

Successive approximations

Function bot always goes into an infinite loop.

What are these?

```
fact0 = \n.(zero? n) 1 (times n (bot   (pred n)));


fact1 = \n.(zero? n) 1 (times n (fact0 (pred n)));


fact2 = \n.(zero? n) 1 (times n (fact1 (pred n)));
```

Successive approximations (manufactured)

```
g = \f.\n.(zero? n) 1 (times n (f (pred n)));

fact0 = g bot;

fact1 = g fact0;          // = g (g bot)

fact2 = g fact1;         // = g (g (g bot))

fact3 = g fact2;         // = g (g (g (g bot)))

   ...
```

Fixed point

Suppose f = g f. I claim f n is n factorial!

Proof by induction on n.

Fixed-point combinator

What if

```
fix g = g (fix g)
```

Then fix g n is n factorial!

```
fix g = g (fix g)
      = g (g (fix g))
      = g (g (g (fix g)))
      = ...
```

Expand as much as you need to.

Y combinator can implement `fix`

Can define Y such that, for any g, $Y\,g = g\;(Y\,g)$.

(Details next time, with evaluation model.)

Conversion to fixed point

```
length = \xs.null? xs 0 (+ 1 (length (cdr xs)))
```

```
lg = \lf.\xs.null? xs 0 (+ 1 (lf (cdr xs)))
```

Example recursion equations

Is there a solution? Is it unique? If so, what is it?

```
f1 = \n.\m.(eq? n m) n
               (plus n (f1 (succ n) m));

f2 = \n.f2 (isZero? n 100 (pred n));

f3 = \xs.xs nil (\z.\zs.cons 0 (f3 zs));

f4 = \xs.\ys.f4 ys xs;
```

Wait for it...

Example recursion equations

```
f1 = \n.\m.(eq? n m) n
                 (plus n (f1 (succ n) m));
    ; sigma (sum from n to m)

f2 = \n.f2 (isZero? n 100 (pred n));
    ; no unique solution (any constant f2)

f3 = \xs.xs nil (\z.\zs.cons 0 (f3 zs));
    ; map (const 0)

f4 = \xs.\ys. f4 xs ys;
    ; not unique: constant funs, commutative op
```

Church Numerals

Encoding natural numbers as lambda-terms

$$\begin{array}{lcl} \text{zero} & = & \lambda f.\lambda x.x \\ \text{one} & = & \lambda f.\lambda x.f\ x \\ \text{two} & = & \lambda f.\lambda x.f(f\ x) \\ \text{succ} & = & \lambda n.\lambda f.\lambda x.f(n\ f\ x) \\ \text{plus} & = & \lambda n.\lambda m.n\ \text{succ}\ m \\ \text{times} & = & \lambda n.\lambda m.n\ (\text{plus}\ m)\ \text{zero} \end{array}$$

Idea: “apply f to x, n times”

Church Numerals to machine integers

```
; uscheme or possibly uhaskell
-> (val add1 ((curry +) 1))
-> (define to-int (n)
             ((n add1) 0))
-> (to-int three)
3
-> (to-int ((times three) four))
12
```

Church Numerals in λ

```
zero  = \f.\x.x;
succ  = \n.\f.\x.f (n f x);
plus  = \n.\m.n succ m;
times = \n.\m.n (plus m) zero;
 ...
-> four;
\f.\x.f (f (f (f x)))
-> three;
\f.\x.f (f (f x))
-> times four three;
\f.\x.f (f (f (f (f (f (f (f (f (f (f (f x)))))))))))
```