PROGRAMMING WITH NOTHING

AN (BRIEF) INTRO TO THE λ-CALCULUS

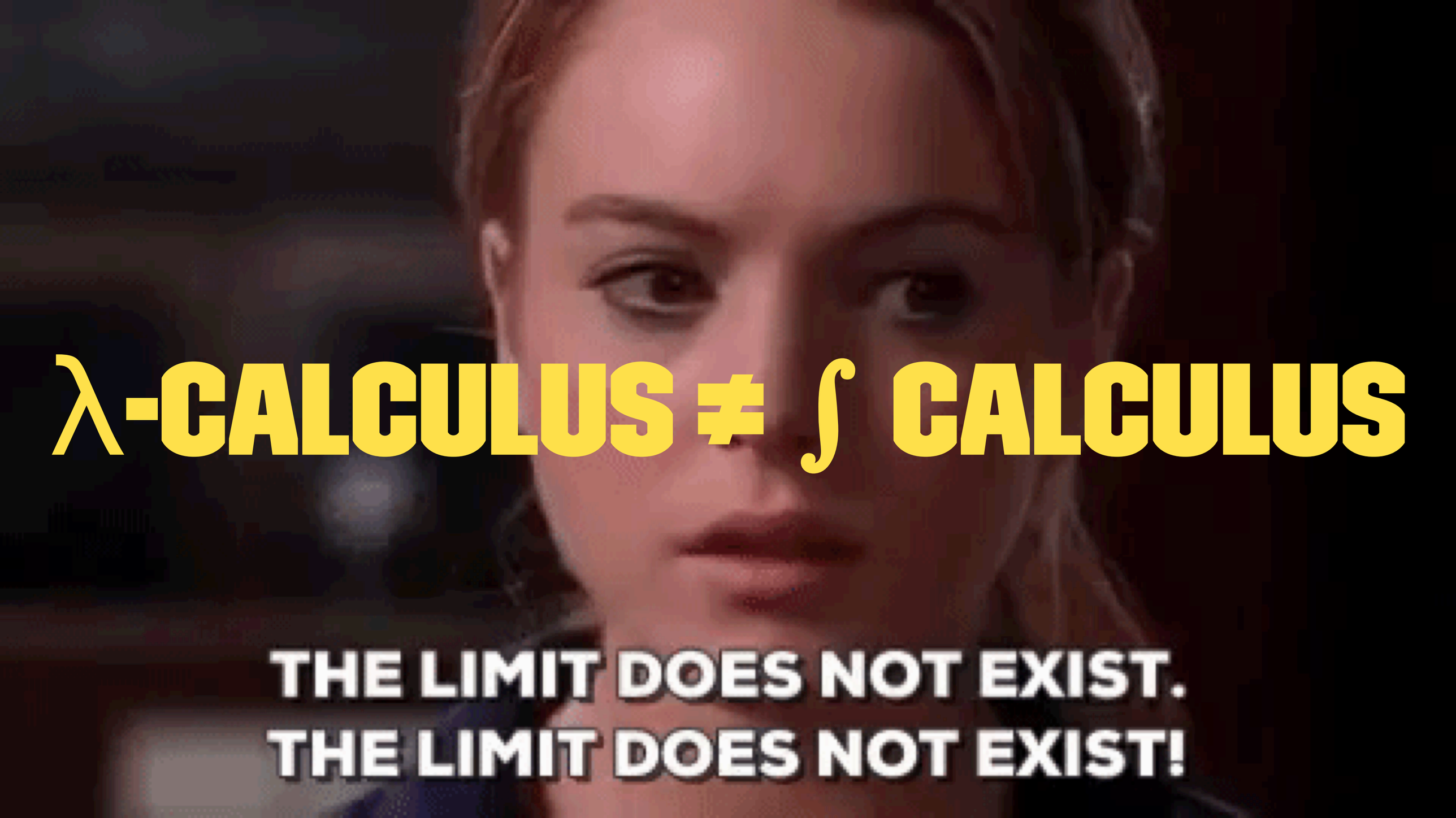

WHALE HELLO THERE

I'm Professor Michael Selsky

iOS Developer on MNE NextGen

@MichaelSelsky

λ-CALCULUS


λ-CALCULUS ≠ ∫ CALCULUS
THE LIMIT DOES NOT EXIST.
THE LIMIT DOES NOT EXIST!

λ-CALCULUS

λ-CALCULUS

» Turing complete

λ-CALCULUS

» Turing complete

» Based entirely around functions

λ-CALCULUS

» Turing complete

» Based entirely around functions

» Invented in the 1930s, before the Turing Machine

ALONZO CHURCH

ALONZO CHURCH

» Invented it to try to solve the Entscheidungsproblem

ALONZO CHURCH

» Invented it to try to solve the Entscheidungsproblem

» Published a paper proving it to be impossible

While λ-calculus is really cool, it can be difficult to read so instead we'll use

DISCLAIMERS

DISCLAIMERS

1. This is not software engineering advice

DISCLAIMERS

1. This is not software engineering advice
2. Don't do this in production

DISCLAIMERS

1. This is not software engineering advice
2. Don't do this in production
3. No seriously, I'll reject that PR

DISCLAIMERS

1. This is not software engineering advice
2. Don't do this in production
3. No seriously, I'll reject that PR
4. This is a learning exercise and a game

DISCLAIMERS

1. This is not software engineering advice
2. Don't do this in production
3. No seriously, I'll reject that PR
4. This is a learning exercise and a game
5. I am not an expert at this


GAME MEANS RULES
THERE ARE CERTAIN RULES
ONE MUST ABIDE BY

RULES

RULES

1. You can create functions

RULES

1. You can create functions
2. You can call functions

RULES

1. You can create functions
2. You can call functions
3. Functions can only take one argument and return one value

RULES

1. You can create functions
2. You can call functions
3. Functions can only take one argument and return one value
4. Can't use the def keyword

RULES

1. You can create functions
2. You can call functions
3. Functions can only take one argument and return one value
4. Can't use the def keyword
5. That's it

THAT'S IT

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

print(fact(6))
```

720

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

print(fact(6))
```

```
ONE = 1

def fact(n):
    if n == 0:
        return ONE
    else:
        return n * fact(n-1)

print(fact(6))
```

```
ONE = 1
IS_ZERO = lambda x: x == 0

def fact(n):
    if IS_ZERO(n):
        return ONE
    else:
        return n * fact(n-1)

print(fact(6))
```

```
fst = λx. λy. x
snd = λx. λy. y
```

```
ONE = 1
IS_ZERO = lambda x: x == 0

def fact(n):
    if IS_ZERO(n):
        return ONE
    else:
        return n * fact(n-1)

print(fact(6))
```

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1

def fact(n):
    if IS_ZERO(n):
        return ONE
    else:
        return n * fact(SUB_1(n))

print(fact(6))
```

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x, y: x * y

def fact(n):
    if IS_ZERO(n):
        return ONE
    else:
        return MULT(n, fact(SUB_1(n)))

print(fact(6))
```


720

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x, y: x * y

def fact(n):
    if IS_ZERO(n):
        return ONE
    else:
        return MULT(n, fact(SUB_1(n)))

print(fact(6))
```

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x, y: x * y
IF = lambda cond, t_val, f_val: t_val if cond else f_val

def fact(n):
    return IF(IS_ZERO(n),
              ONE,
              MULT(n, fact(SUB_1(n))))

print(fact(6))
```


RECURSIONERROR: MAXIMUM
RECURSION DEPTH EXCEEDED
IN COMPARISON

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x, y: x * y
IF = lambda cond, t_val, f_val: t_val if cond else f_val

def fact(n):
    return IF(IS_ZERO(n),
              ONE,
              MULT(n, fact(SUB_1(n))))

print(fact(6))
```

But, we can fix this

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x, y: x * y
IF = lambda cond, t_func, f_func: t_func(None) if cond else f_func(None)

def fact(n):
    return IF(IS_ZERO(n),
              lambda _: ONE,
              lambda _: MULT(n, fact(SUB_1(n))))

print(fact(6))
```

720

ノ(゜-゜)ノ

```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x, y: x * y
IF = lambda cond, t_func, f_func: t_func(None) if cond else f_func(None)

def fact(n):
    return IF(IS_ZERO(n),
              lambda _: ONE,
              lambda _: MULT(n, fact(SUB_1(n))))

print(fact(6))
```


LET'S GET RID OF THAT
DEF

```
fact = lambda n:
           IF(IS_ZERO(n),
                lambda _: ONE,
                lambda _: MULT(n, fact(SUB_1(n))))

print(fact(6))
```

Except we can't use a name in it's own definition.
That's just weird.

So we can just pass it in instead

```
fact =  lambda fact, n:
          IF(IS_ZERO(n),
              lambda _: ONE,
              lambda _: MULT(n, fact(fact, SUB_1(n))))

print(fact(fact, 6))
```

```
fact =  lambda myself, n:
            IF(IS_ZERO(n),
              lambda _: ONE,
              lambda _: MULT(n, myself(myself, SUB_1(n))))

print(fact(fact, 6))
```

REMEMBER RULE 3

RULE 3

FUNCTIONS CAN ONLY TAKE ONE ARGUMENT

```
MULT = lambda x, y: x * y
IF = lambda cond, t_func, f_func: t_func(None) if cond else f_func(None)
fact =  lambda myself, n: ...
```

These take 2 or 3 arguments


But, we can fix this

HIGHER ORDER FUNCTIONS

Functions that take and/or return other functions


MATH

$$\lim_{x \to \infty} f(x)$$

$$\int_a^b x^2\,dx$$

$$\sum_{n=1}^{\infty} 2^{-n}$$

FUNCTIONAL PROGRAMMING

```
[1, 2, 3, 4].map(square) # => [1, 4, 9, 16]
```

```
[1, 2, 3, 4].filter(isEven) # => [2, 4]
```

```
[1, 2, 3, 4].reduce(0, +) # => 10
```


CURRYING

SCHÖNFINKELISATION

“TECHNIQUE OF TRANSLATING THE EVALUATION OF A FUNCTION THAT TAKES MULTIPLE ARGUMENTS INTO EVALUATING A SEQUENCE OF FUNCTIONS, EACH WITH A SINGLE ARGUMENT.”

Wikipedia

HASKELL CURRY

HASKELL CURRY

» Traced back to 1893

HASKELL CURRY

» Traced back to 1893

» Curry-Howard correspondence

```
add = lambda x, y = x + y
add(2, 3) # => 5

add = lambda x: lambda y: x + y
add(2)(3) # => 5

add2 = add(2)
add2(6) # => 8
```

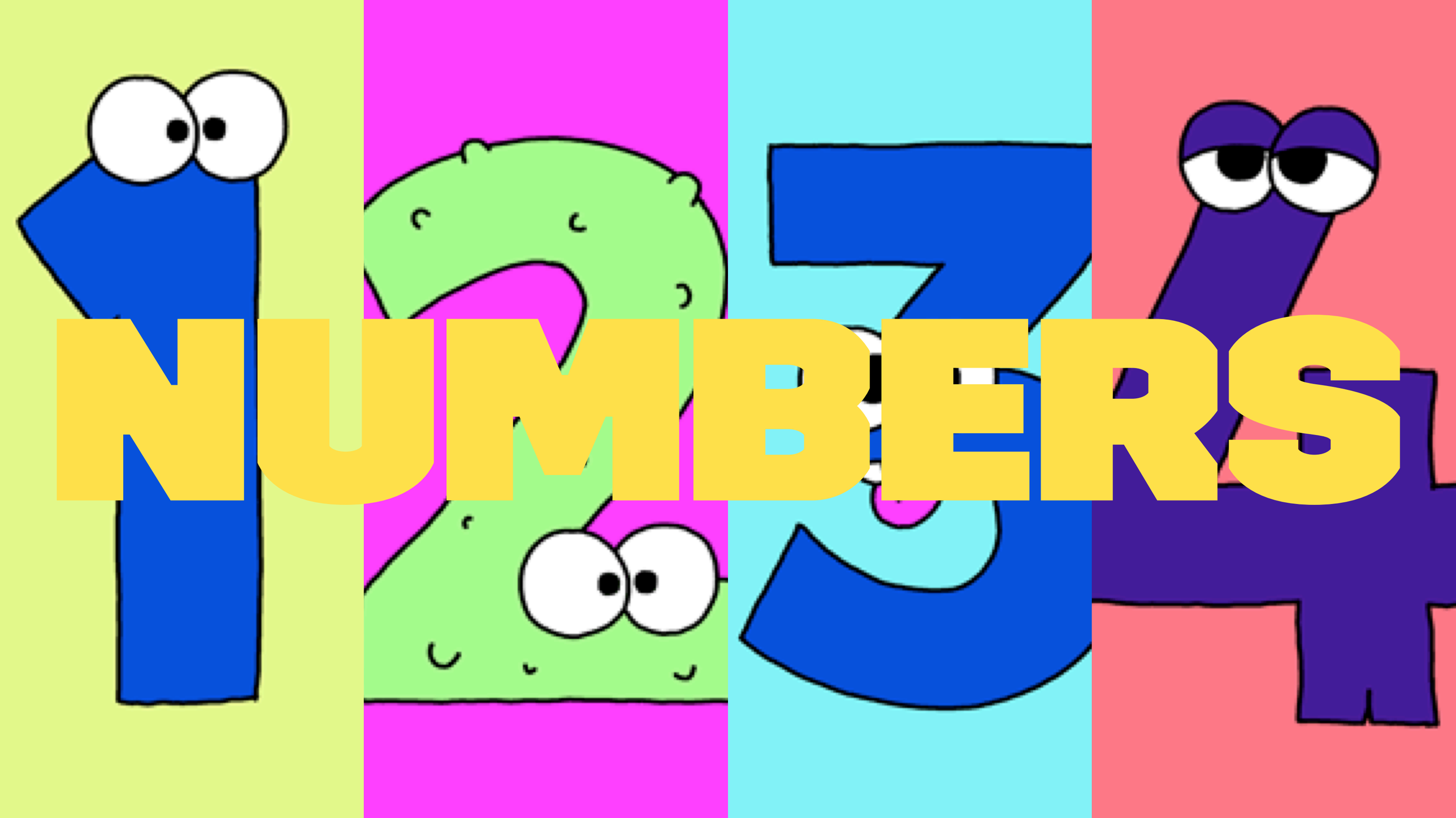
```
ONE = 1
IS_ZERO = lambda x: x == 0
SUB_1 = lambda x: x - 1
MULT = lambda x: lambda y: x * y
IF = lambda cond: lambda t_func: lambda f_func: f_func: t_func(None) if cond else f_func(None)

fact =  lambda myself: lambda n:
      IF(
        IS_ZERO(n)
      )(
        lambda _: ONE
      )(
        lambda _: MULT(n)(myself(myself)(SUB_1(n)))
      )

print(fact(fact)(6))
```

COOL, SO EVERYTHING IS A FUNCTION

EXCEPT

```
ONE = 1
```


But, we can fix this


NUMBERS

Alonzo Church figured this out for us

CHURCH NUMERALS

CHURCH NUMERALS

CHURCH NUMERALS

» Just another way to represent positive integers

CHURCH NUMERALS

» Just another way to represent positive integers

» Think "Roman Numerals" but with functions

CHURCH NUMERALS

» Just another way to represent positive integers

» Think "Roman Numerals" but with functions

» Work by applying the same function over and over

A number n is defined as the function f applied n times to a based value of x

```
0 = x
1 = f(x)
2 = f(f(x))
3 = f(f(f(x)))
...
n = f(f(f(... n times ...(x))))
```

```
ZERO = lambda f: lambda x: x
ONE = lambda f: lambda x: f(x)
TWO = lambda f: lambda x: f(f(x))
```

CONVERSION

```
TWO(lambda x: x + 1)(0) # => 2
```

```
THREE(lambda: x * 2)(1) # => 8
```

ADDITION

So let's build this up through a few simple cases

SIMPLEST CASE

ADDING 0
AKA IDENTITY

```
IDENTITY = lambda n: lambda f: lambda x: n(f)(x)
```

```
IDENTITY = lambda n: lambda f: lambda x: n(f)(x)
```

» Take in a number n and return a function that takes in a f and x and then passes those into the number n

```
IDENTITY = lambda n: lambda f: lambda x: n(f)(x)
```

» Take in a number n and return a function that takes in a f and x and then passes those into the number n

» We can think of any function that takes in a f and x as a number

SIMPLE CASE

ADD 1
AKA SUCCESSOR

Remember what how we define a number

A number n is defined as the function f applied n times to a based value of x

So adding one is just calling f one more time

```
SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
```

```
SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
```

» Take in a number n and return another number

» lambda f: lambda x:

```
SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
```

» Take in a number n and return another number

  » lambda f: lambda x:

» Apply f one more time to number n

```
TWO = lambda f: lambda x: f(f(x))

SUCC = lambda n: lambda f: lambda x: f(n(f)(x))

THREE = SUCC(TWO) # => f(f(f(n)))
```

ADDITION

Remember what how we define a number

A number n is defined as the function f applied n times to a based value of x

```
m + n = m(SUCC)(n)

We apply add 1 m times to n

ADD = lambda n: lambda m: n(SUCC)(m)
```

We're also going to need predecessor aka SUB_1

```
PRED = lambda n:
  lambda f: lambda x:
    n(lambda g: lambda h: h(g(f)))
    (lambda u: x)(lambda u: u)
```

This applies f one less time.

It's a little too confusing to explain.

```
MULT = (
        lambda n: lambda m: n(lambda x:
                              ADD(x)(m)) (ZERO)
    )
```


AND THAT'S
NUMBERS

That will be on the quiz

That will be on the quiz

Another data type

BOOLEANS

So what are booleans?


BOOLEANS = CHOICE
ON THE ONE
HAND...

```
NIL = lambda x: x
TRUE = lambda t_func: lambda f_func: t_func(NIL)
FALSE = lambda t_func: lambda f_func: f_func(NIL)

IF = lambda cond: lambda t_func: lambda f_func:
    cond(t_func)(f_func)

IS_ZERO = lambda n: n(lambda x: FALSE)(TRUE)
```


NOW WE HAVE ALL THE
PIECES TO DO THIS

```
NULL = lambda x: x
TRUE = lambda t: lambda f: t(NULL)
FALSE = lambda t: lambda f: f(NULL)
IF = lambda cond: lambda t: lambda f: cond(t)(f)

ZERO = lambda f: lambda x: x
ADD1 = lambda n: (lambda f: lambda x: f(n (f)(x)))
ADD = lambda n: lambda m: n(ADD1)(m)
ONE = ADD1(ZERO)
IS_ZERO = lambda n: n(lambda x: FALSE)(TRUE)
SUB1 = (
  (lambda n:
   lambda f:
   lambda x: n (lambda g: lambda h: h(g(f)))
               (lambda u: x)
               (lambda u: u))
)
MULT = (
  lambda n: lambda m: n(lambda x:
                        ADD(x)(m)) (ZERO)
)

SIX = ADD1(ADD1(ADD1(ADD1(ADD1(ADD1(ZERO))))))
```

```
FACT = (
        lambda myself: (
            lambda n: (
                IF(
                    IS_ZERO(n)
                )(
                    lambda _: ONE
                )(
                    lambda _: MULT(n)(myself(myself)(SUB1(n)))
                )
            )
        )
    )

print(FACT(FACT)(SIX))
```

```
<function __main__.<lambda>.<locals>.<lambda>>
```

Cool it works

```
print(FACT(FACT)(SIX)(lambda x: x + 1)(0))
```

720

```
def fact(n):
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

print(fact(6))
```

```
NULL = lambda x: x
TRUE = lambda t: lambda f: t(NULL)
FALSE = lambda t: lambda f: f(NULL)
IF = lambda cond: lambda t: lambda f: cond(t)(f)

ZERO = lambda f: lambda x: x
ADD1 = lambda n: (lambda f: lambda x: f(n (f)(x)))
ADD = lambda n: lambda m: n(ADD1)(m)
ONE = ADD1(ZERO)
IS_ZERO = lambda n: n(lambda x: FALSE)(TRUE)
SUB1 = (
  (lambda n:
   lambda f:
   lambda x: n (lambda g: lambda h: h(g(f)))
               (lambda u: x)
               (lambda u: u))
)
MULT = (
  lambda n: lambda m: n(lambda x:
                        ADD(x)(m)) (ZERO)
)

SIX = ADD1(ADD1(ADD1(ADD1(ADD1(ADD1(ZERO))))))

FACT = (
          lambda myself: (
              lambda n: (
                  IF(
                      IS_ZERO(n)
                  )(
                      lambda _: ONE
                  )(
                      lambda _: MULT(n)(myself(myself)(SUB1(n)))
                  )
              )
          )
      )

print(FACT(FACT)(SIX))
```

So we're done

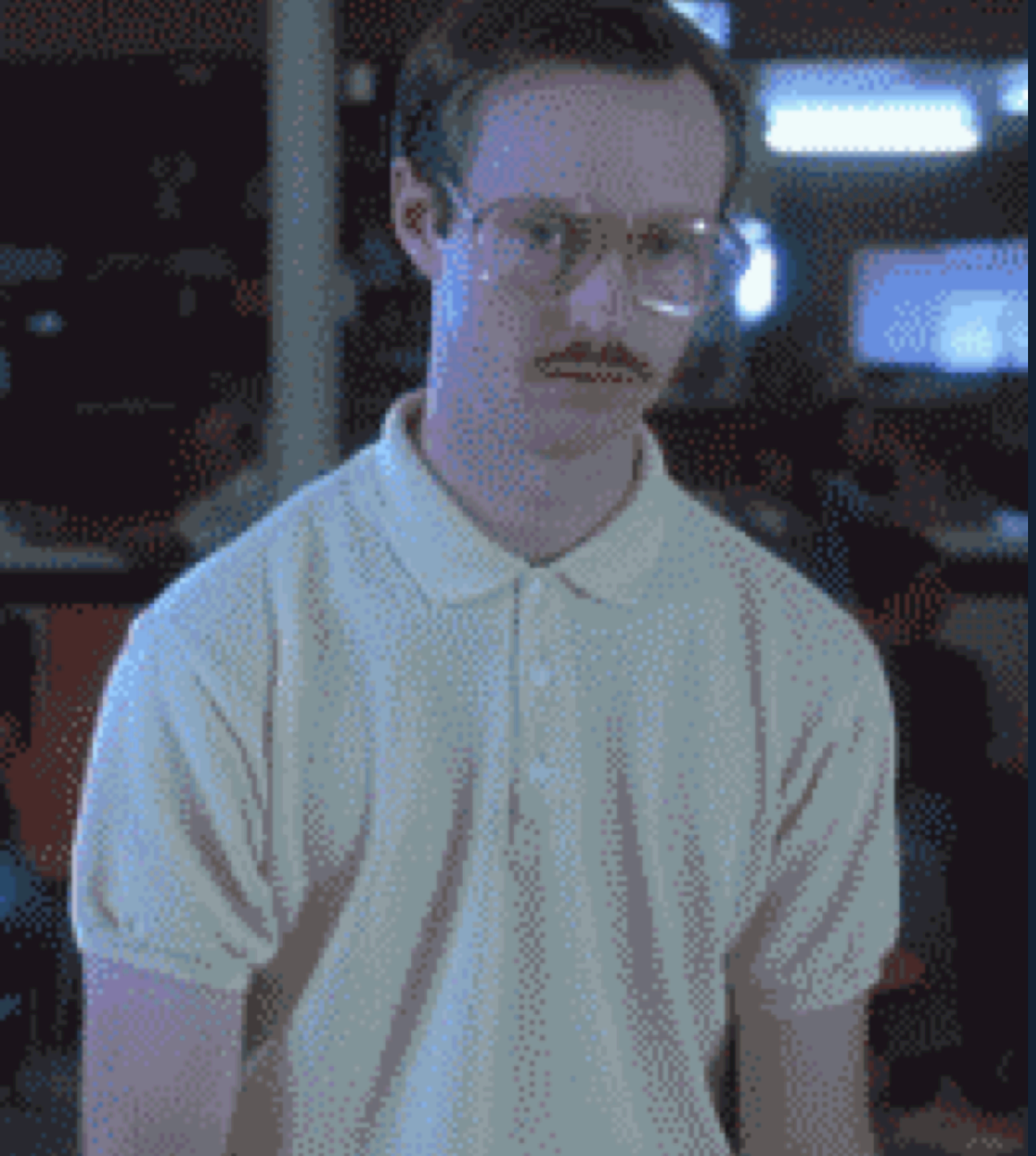
EXCEPT

```
NULL = lambda x: x
TRUE = lambda t: lambda f: t(NULL)
FALSE = lambda t: lambda f: f(NULL)
IF = lambda cond: lambda t: lambda f: cond(t)(f)

ZERO = lambda f: lambda x: x
ADD1 = lambda n: (lambda f: lambda x: f(n (f)(x)))
ADD = lambda n: lambda m: n(ADD1)(m)
ONE = ADD1(ZERO)
IS_ZERO = lambda n: n(lambda x: FALSE)(TRUE)
SUB1 = (
  (lambda n:
   lambda f:
   lambda x: n (lambda g: lambda h: h(g(f)))
               (lambda u: x)
               (lambda u: u))
)
MULT = (
  lambda n: lambda m: n(lambda x:
                        ADD(x)(m)) (ZERO)
)

SIX = ADD1(ADD1(ADD1(ADD1(ADD1(ADD1(ZERO))))))

FACT = (
```

But, we can fix this

```
NULL = lambda x: x
TRUE = lambda t: lambda f: t(NULL)
FALSE = lambda t: lambda f: f(NULL)
IF = lambda cond: lambda t: lambda f: cond(t)(f)

ZERO = lambda f: lambda x: x
ADD1 = lambda n: (lambda f: lambda x: f(n (f)(x)))
ADD = lambda n: lambda m: n(ADD1)(m)
ONE = ADD1(ZERO)
IS_ZERO = lambda n: n(lambda x: FALSE)(TRUE)
SUB1 = (
  (lambda n:
   lambda f:
   lambda x: n (lambda g: lambda h: h(g(f)))
               (lambda u: x)
               (lambda u: u))
)
MULT = (
  lambda n: lambda m: n(lambda x:
                        ADD(x)(m)) (ZERO)
)

SIX = ADD1(ADD1(ADD1(ADD1(ADD1(ADD1(ZERO))))))

FACT = (
```

```
print(
      (lambda myself: (lambda n: ((lambda cond: lambda t: lambda f: cond(t)(f))((lambda n: n(lambda x: lambda t: lambda f: f(lambda x:
      x))(lambda t: lambda f: t(lambda x: x)))(n))(lambda _: (lambda n: (lambda f: lambda x: f(n (f)(x))))(lambda f: lambda x:
      x))(lambda _: (lambda n: lambda m: n(lambda x:(lambda n: lambda m: n(lambda n: (lambda f: lambda x: f(n (f)(x))))(m))(x)(m))
      (lambda f: lambda x: x))(n)(myself(myself)(((lambda n:lambda f:lambda x: n (lambda g: lambda h: h(g(f)))(lambda u: x)(lambda u:
      u)))(n)))))))(lambda myself: (lambda n: ((lambda cond: lambda t: lambda f: cond(t)(f))((lambda n: n(lambda x: lambda t: lambda
      f: f(lambda x: x))(lambda t: lambda f: t(lambda x: x)))(n))(lambda _: (lambda n: (lambda f: lambda x: f(n (f)(x))))(lambda f:
      lambda x: x))(lambda _: (lambda n: lambda m: n(lambda x:(lambda n: lambda m: n(lambda n: (lambda f: lambda x: f(n
      (f)(x))))(m))(x)(m)) (lambda f: lambda x: x))(n)(myself(myself)(((lambda n:lambda f:lambda x: n (lambda g: lambda h:
      h(g(f)))(lambda u: x)(lambda u: u)))(n)))))))( (lambda n: (lambda f: lambda x: f(n (f)(x))))((lambda n: (lambda f: lambda
      x: f(n (f)(x))))((lambda n: (lambda f: lambda x: f(n (f)(x))))((lambda n: (lambda f: lambda x: f(n (f)(x))))((lambda n:
      (lambda f: lambda x: f(n (f)(x))))((lambda n: (lambda f: lambda x: f(n (f)(x))))(lambda f: lambda x: x)))))))

  (lambda x: x + 1)(0)
)
```



720


"SO WHY THE F**K WOULD YOU DO THIS?"
You, the audience member
WHYYYYY

1. IT'S FUN

2. IT GIVES PERSPECTIVE

Like isn't it cool how functions can represent the basic building blocks of programming

3. THIS IS WHERE FUNCTIONAL PROGRAMMING STEMS FROM

4. A SIMPLE ALTERNATIVE TO TURING MACHINES

THANKS A BUNCH!


Any questions?
NBC