Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434814_24436.lc)

```
let TRIPLE = \x y z -> ???
let FST3   = \t -> ???
let SND3   = \t -> ???
let THD3   = \t -> ???

eval ex1:
  FST3 (TRIPLE apple banana orange)
  =*> apple

eval ex2:
  SND3 (TRIPLE apple banana orange)
  =*> banana

eval ex3:
  THD3 (TRIPLE apple banana orange)
  =*> orange
```

I mean this

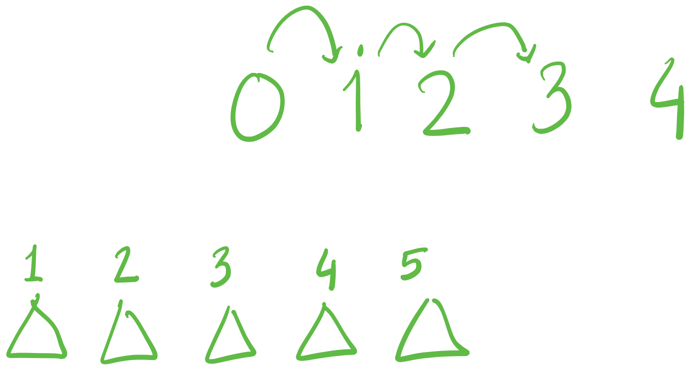
$\boxed{\text{Closed}}$

NO FREE VARS

Programming in λ-calculus

- **Booleans** [done]
- **Records** (structs, tuples) [done]
- Numbers
- **Functions** [we got those]

- Recursion



λ-calculus: Numbers

Let's start with **natural numbers** (0, 1, 2, …)

What do we *do* with natural numbers?

- Count: `0` , `inc`
- Arithmetic: `dec` , `+` , `-` , `*`
- Comparisons: `==` , `<=` , etc

Natural Numbers: API

We need to define:

COUNT

- A family of **numerals**: `ZERO`, `ONE`, `TWO`, `THREE`, ...
- Arithmetic functions: `INC`, `DEC`, `ADD`, `SUB`, `MULT`
- Comparisons: `IS_ZERO`, `EQ`

LSS

Such that they respect all regular laws of arithmetic, e.g.

```
IS_ZERO ZERO       =~> TRUE
IS_ZERO (INC ZERO) =~> FALSE
INC ONE            =~> TWO
...
```

INC TWO => THREE

Natural Numbers: Implementation

Church numerals: *a number* N is encoded as a combinator that *calls a function on an argument* N *times*

```
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (f (f (f x)))))
...
```

$$n = \backslash f\ x \rightarrow \underbrace{f \cdots (f}_{n \text{ times}}\ x)$$

QUIZ: Church Numerals

Which of these is a valid encoding of `ZERO` ?

- **A**: `let ZERO = \f x -> x`
- **B**: `let ZERO = \f x -> f`

$n = \backslash f\ x \to \underbrace{f \cdots (f}_{n \text{ times}}\ x)$

$\backslash f\ x \to f\ (f\ x)$

- **C**: `let ZERO = \f x -> f x`
- **D**: `let ZERO = \x -> x`
- **E**: None of the above

Does this function look familiar?

λ-calculus: Increment

```
-- Call `f` on `x` one more time than `n` does
let INC   = \n -> (\f x -> ???)
```

f (call_f_on_x_ntimes)

takes some g, y

\g y -> g (g (g .... y))

n-times

f ntimes

n+1

n f x

=*> $\underbrace{f \ldots (f (f (f}_{n}$ x)))

let INC = \n -> (\f x -> f (n f x))

let INCU = \n -> (\f x -> n f (f x))

Example:

```
eval inc_zero :
  INC ZERO
  =d> (\n f x -> f (n f x)) ZERO
  =b> \f x -> f (ZERO f x)
  =*> \f x -> f x
  =d> ONE
```

EXERCISE

Fill in the implementation of `ADD` so that you get the following behavior

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585436042_24449.lc)

```
let ZERO = \f x -> x
let ONE  = \f x -> f x
let TWO  = \f x -> f (f x)
let INC  = \n f x -> f (n f x)

let ADD  = fill_this_in

eval add_zero_zero:
  ADD ZERO ZERO =~> ZERO

eval add_zero_one:
  ADD ZERO ONE =~> ONE

eval add_zero_two:
  ADD ZERO TWO =~> TWO

eval add_one_zero:
  ADD ONE ZERO =~> ONE
```

```
eval add_one_zero:
  ADD ONE ONE =~> TWO

eval add_two_zero:
  ADD TWO ZERO =~> TWO
```

QUIZ

$n\ f\ x \Rightarrow \underbrace{f(f.f}_{n}..x))$

How shall we implement `ADD` ?

A. **let** `ADD = \n m -> n INC m` ✓ $\underbrace{INC...(INC(INC}_{n}\ m)$

m INC n

B. **let** `ADD = \n m -> INC n m`

2 args ✗

C. **let** `ADD = \n m -> n m INC`

$\underbrace{m.m}_{n}(m\ INC)$

D. **let** `ADD = \n m -> n (m INC)`

E. **let** `ADD = \n m -> n (INC m)`

$n\ (m+1)$

λ-calculus: Addition

```
--  Call `f` on `x` exactly `n + m` times
let ADD = \n m -> n INC m
```

Example:

```
eval add_one_zero :
  ADD ONE ZERO
  =~> ONE
```

QUIZ

MUL 3 4

$\boxed{4}\ \boxed{4}\ \boxed{4}$ 3 times

How shall we implement `MULT`?

wrong args to ADD

A. **let** MULT = \n m -> n ADD m

B. **let** MULT = \n m -> n (ADD m) ZERO

C. **let** MULT = \n m -> m (ADD n) ZERO

D. **let** MULT = \n m -> n (~~ADD~~ m ~~ZERO~~) m

E. **let** MULT = \n m -> (n ADD m) ZERO

ADD (ADD ... ADD (m)) n

ADD m ... (ADD m) (ADD m ZERO))) n

ADD n ... (ADD n ... (ZERO)) m

n = loop f on x "n" times

base = m

loop_n:

bs = add m base

λ-calculus: *Multiplication*

```
--  Call `f` on `x` exactly `n * m` times
let MULT = \n m -> n (ADD m) ZERO
```

Example:

```
eval two_times_three :
  MULT TWO ONE
  =~> TWO
```

Programming in λ-calculus

- **Booleans** [done]
- **Records** (structs, tuples) [done]
- **Numbers** [done]
- **Functions** [we got those]
- Recursion

n $\quad\quad$ $\backslash f\ x \to \underbrace{f \ldots (f}_{n\text{-copies}}\ x)$