/elsa/index.html#?demo=permalink%2F1585436042_24449.lc)

$$n\ f\ x \equiv \underbrace{f \cdots (f}_{n \text{ times}}\ x)$$

\n -> \f x -> n f (f x)

```
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
```

Comprehensive

02- WHILE

03- TRANSFORMERS

$\geq 80\%$

QUIZ

How shall we implement `ADD` ?

(INC ....INC (INC m))

A. `let ADD = \n m -> n INC m`

n

B. `let ADD = \n m -> (INC n) m` (n+1) m

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

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

"m+1"

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

$\overbrace{(\text{ADD} \ldots \text{ADD}}^{n}\ (\text{ADD}\ m))$

How shall we implement `MULT` ?

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

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

$\underbrace{(\text{ADD}\ m \ldots (\text{ADD}\ m\ (\text{ADD}\ m\ \text{ZERO})))}_{n\text{-times}}$

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

$\underbrace{m + m + m \ldots + m}_{n\text{-times}}$

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

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

F. $\backslash n\ m \to (\backslash f\ x \to n\ (m\ f)\ x)$

$(m\ f \ldots (m\ f\ (m\ f\ (m\ f\ x))))$

n

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]
- **Lists**
- **Functions** [we got those]
- Recursion

λ-calculus: Lists

Lets define an API to build lists in the λ-calculus.

An Empty List

```
NIL
```

Constructing a list

A list with 4 elements

```
CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

intuitively `CONS h t` creates a *new* list with

- *head* `h`
- *tail* `t`

Destructing a list

- `HEAD l` returns the *first* element of the list
- `TAIL l` returns the *rest* of the list

```
HEAD (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL))))
=~> apple
```

HEAD

```
TAIL (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL))))
```

~~=~> CONS banana (CONS cantaloupe (CONS dragon NIL)))~~

=~> banana

λ-calculus: Lists

```
let NIL  = ???
let CONS = ???
let HEAD = ???
let TAIL = ???

eval exHd:
  HEAD (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL))))
  =~> apple

eval exTl
  TAIL (CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL))))
  =~> CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

EXERCISE: Nth

Write an implementation of `GetNth` such that

- `GetNth n l` returns the `n`-th element of the list `l`

Assume that `l` has n or more elements

```
let GetNth = ???

eval nth1 :
  GetNth ZERO (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> apple

eval nth1 :
  GetNth ONE (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> banana

eval nth2 :
  GetNth TWO (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> cantaloupe
```

Click here to try this in elsa (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1586466816_52273.lc)

λ-calculus: Recursion

let DEC = \n -> . . .

I want to write a function that sums up natural numbers up to `n` :

```
let SUM = \n -> ...  -- 0 + 1 + 2 + ... + n
```

def sum (n):
  i = 0
  r = 0
  repeat n times:
    r += i
    i += 1

such that we get the following behavior

```
eval exSum0: SUM ZERO  =~> ZERO
eval exSum1: SUM ONE   =~> ONE
eval exSum2: SUM TWO   =~> THREE
eval exSum3: SUM THREE =~> SIX
```

0+1+2+3

(0, 0) → (1, 0+0) → (2, 1) → (3, 3)
i, r   i+1, r+1

Can we write sum **using Church Numerals**?

sum

n

Click here to try this in Elsa (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1586465192_52175.lc)

QUIZ

You *can* write `SUM` using numerals but its *tedious*.

Is this a correct implementation of `SUM`?

```
let SUM = \n -> ITE (ISZ n)
            ZERO
            (ADD n (SUM (DEC n)))
```

A. Yes

B. No

No!

- Named terms in Elsa are just syntactic sugar
- To translate an Elsa term to λ-calculus: replace each name with its definition

```
\n -> ITE (ISZ n)
        ZERO
        (ADD n (SUM (DEC n))) -- But SUM is not yet defined!
```

Recursion:

- Inside *this* function
- Want to call the *same* function on `DEC n`

Looks like we can't do recursion!

- Requires being able to refer to functions *by name*,
- But λ-calculus functions are *anonymous*.

Right?

λ-calculus: Recursion

Think again!

Recursion:

Instead of

- ~~Inside *this* function I want to call the *same* function on `DEC n`~~

Lets try

- Inside *this* function I want to call *some* function `rec` on `DEC n`
- And BTW, I want `rec` to be the *same* function

Step 1: Pass in the function to call "recursively"

```
let STEP =
  \rec -> \n -> ITE (ISZ n)
                    ZERO
                    (ADD n (rec (DEC n))) -- Call some rec
```

Step 2: Do some magic to `STEP`, so `rec` is itself

```
\n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))
```

That is, obtain a term `MAGIC` such that

```
MAGIC =*> STEP MAGIC
```

MAGIC n =*> STEP MAG n

λ-calculus: Fixpoint Combinator

Wanted: a λ-term `FIX` such that

- `FIX STEP` calls `STEP` with `FIX STEP` as the first argument:

```
(FIX STEP) =*> STEP (FIX STEP)
```

(In math: a *fixpoint* of a function $f(x)$ is a point x, such that $f(x) = x$)

Once we have it, we can define:

```
let SUM = FIX STEP
```

Then by property of `FIX` we have:

```
SUM   =*>  FIX STEP  =*>  STEP (FIX STEP)  =*>  STEP SUM
```

and so now we compute:

```
eval sum_two:
  SUM TWO
  =*> STEP SUM TWO
  =*> ITE (ISZ TWO) ZERO (ADD TWO (SUM (DEC TWO)))
  =*> ADD TWO (SUM (DEC TWO))
  =*> ADD TWO (SUM ONE)
  =*> ADD TWO (STEP SUM ONE)
  =*> ADD TWO (ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE))))
  =*> ADD TWO (ADD ONE (SUM (DEC ONE)))
  =*> ADD TWO (ADD ONE (SUM ZERO))
  =*> ADD TWO (ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM DEC ZER
O)))
  =*> ADD TWO (ADD ONE (ZERO))
  =*> THREE
```

How should we define `FIX` ???

The Y combinator

Remember Ω?

```
(\x -> x x) (\x -> x x)
=b> (\x -> x x) (\x -> x x)
```

This is *self-replcating code*! We need something like this but a bit more involved...

The Y combinator discovered by Haskell Curry:

```
let FIX   = \stp -> (\x -> stp (x x)) (\x -> stp (x x))
```

How does it work?

let FSTEP = \f n -> ISZ n ONE (MUL n (f (DEC n))) let FAC = FIX FSTEP

Y_k FSTEP

```
eval fix_step:
  FIX STEP
  =d> (\stp -> (\x -> stp (x x)) (\x -> stp (x x))) STEP
  =b> (\x -> STEP (x x)) (\x -> STEP (x x))
  =b> STEP ((\x -> STEP (x x)) (\x -> STEP (x x)))
  --       ^^^^^^^^^^ this is FIX STEP ^^^^^^^^^^^
```

That's all folks, Haskell Curry was very clever.

Next week: We'll look at the language named after him (`Haskell`)

(https://ucsd-cse230.github.io/fa21/feed.xml) (https://twitter.com/ranjitjhala) (https://plus.google.com/u/0/104385825850161331469) (https://github.com/ranjitjhala)