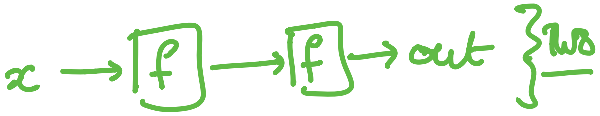
Natural Numbers: Implementation

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

let zero = \f x → x

```
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)))))
...
```



$x \to$ out } zero

"0 copies of f"

let N = \f x → $\underbrace{f \ldots f (f (f}_{N\text{-times}}$ 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`
- **C**: `let ZERO = \f x -> f x`
- **D**: `let ZERO = \x -> x`
- **E**: None of the above

$$\text{let } \underline{N} = \backslash \underline{f}\ \underline{x} \to \underbrace{f \ldots (f\,(f\ x))}_{N\text{-times}}$$

Does this function look familiar?

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

λ-calculus: Increment

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

"n+1"

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

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

" the λ-term corresp to n+1 "

How to "call" f on x exactly "n" times?

$$\boxed{f\ (\underbrace{n\ f\ x}_{})}$$

1 + n- times

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

How shall we implement `ADD` ?

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

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

EXERCISE

ADD = \n m → ???

"n+m"

n m

$$\backslash f\ x \to \underbrace{(f \ldots (f}_{m+n}\ x))$$

INC +1

$$\boxed{\underbrace{(+1 \ldots\ldots +1\ \ +1\ \ +1}_{m\text{-times}}\ n)}$$

m+n

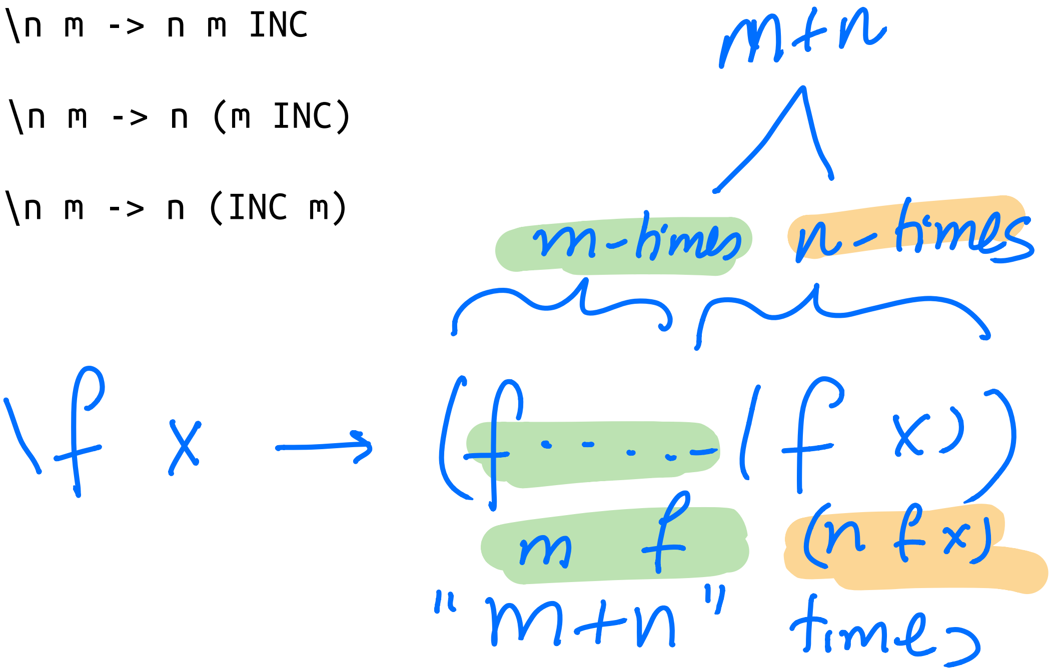
$$\underbrace{+1 \ldots\ldots +1}_{n\text{-times}}\ m$$

n+m

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

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

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



ADD = \n m → (\f x → m f (n f x))

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

MULT TWO ZERO => ZERO
TWO ONE => ONE
TWO TWO => FOUR

How shall we implement `MULT`?

✗ **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

\n m -> n m

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

2

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

$\overbrace{m + \ldots\ m + m + m}^{n\text{-times}} + 0 = n \times m$

$\overbrace{n + \ldots\ \ldots\ n + n}^{m} + 0$

ADD expects 2 args but given 1

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** Datatypes.
- **Functions** [we got those]
- Recursion

λ-calculus: Lists

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

An Empty List

```
NIL
```

'empty' /null

Constructing a list

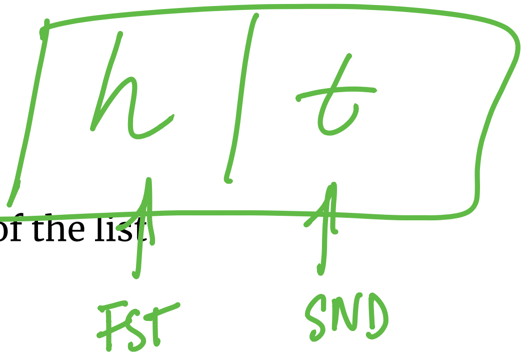
A list with 4 elements

4 3 2 1

```
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 N
IL))))
=~> apple

TAIL (CONS apple (CONS banana (CONS cantaloupe (CONS dragon N
IL))))
=~> CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

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

→ list

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

Assume that `l` has n or more elements

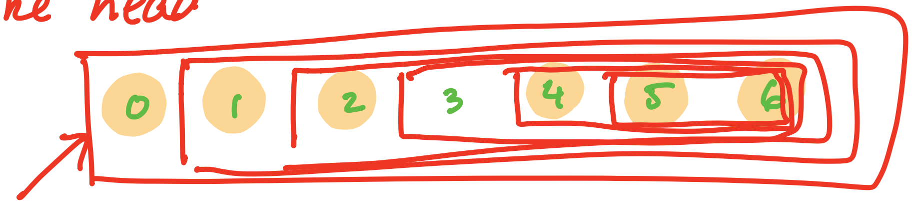
num

```
let GetNth = ???

eval nth1 :
                     0               1               2
  GetNth ZERO (CONS apple (CONS banana (CONS cantaloupe NI
L)))
  =~> apple

eval nth1 :
                    0                1                  2
  GetNth ONE (CONS apple (CONS banana (CONS cantaloupe NIL)))
  =~> banana

eval nth2 :
                    0                1                  2
  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)

→ move "cursor" n times to right "tail"

→ then take "head"



=

head (n tail l)

λ-calculus: Recursion

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

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

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)
(0+1)
(0+1+2)
(0+1+2+3)

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

TRY THIS AT HOME!

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

$e := \backslash x \to e \mid (e_1\ e_2) \mid x$

QUIZ

→ try at home

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

```
def sum(n):
    if n==0:
        return 0
    else:
        retun n+ sum(n-1)
```

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
```

n-1

the func to call recursive

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 ⟹ SUBODY MAGIC

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:

SUM = FIX SUM_BODY

```
let SUM = FIX STEP
```

Then by property of `FIX` we have:

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

||

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 ZERO)))
  =*> 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?

```
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 ^^^^^^^^^^^
```

FixSTEP = STEP (Fix STEP)

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

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

(https://ucsd-cse230.github.io/fa20/feed.xml)
(https://twitter.com/ranjitjhala)
(https://plus.google.com/u/0/104385825850161331469)
(https://github.com/ranjitjhala)

Generated by Hakyll (http://jaspervdj.be/hakyll), template by Armin Ronacher (http://lucumr.pocoo.org), suggest improvements here (https://github.com/ucsd-progsys/liquidhaskell-blog/).