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

eval add one zero:

eval add_two_zero:

ADD ONE ONE =~> TWO

ADD TWO ZERO =~> TWO

Comprehensive 92-WHILE 03-TRANSFORMERS

9/23/21, 12:07 PM 69 of 88

QUIZ

B. let ADD =
$$\n m \rightarrow (INC n) m$$

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

 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

cse230

λ -calculus: Multiplication

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

Example:

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

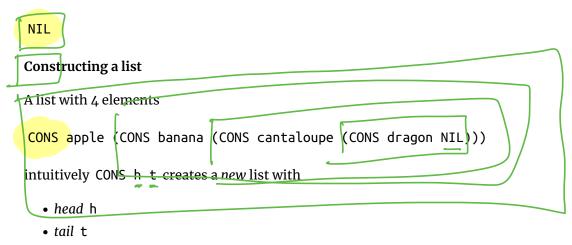
Programming in λ -calculus

- **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



Destructing a list

- HEAD l returns the first element of the list
- TAIL \ returns the rest of the list

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

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

=a) 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)))
```

77 of 88 9/23/21, 12:07 PM

EXERCISE: Nth

Write an implementation of GetNth such that

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

Assume that 1 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

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

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

such that we get the following behavior

def sum (n):

i=0
r=0
repeat n times:
r+=i
i+>1

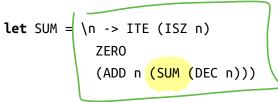
$$(0,0) \longrightarrow (1,0+0) \longrightarrow (2,1) \longrightarrow (3,3)$$

$$i \quad Y \qquad i+1, r+1$$

QUIZ

You can write SUM using numerals but its tedious.

Is this a correct implementation of SUM?



A. Yes

B. No

No!

- Named terms in Elsa are just syntactic sugar
- \bullet To translate an Elsa term to $\lambda\text{-calculus} :$ replace each name with its definition

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"

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 $\,{\rm MAGIC}\,$ such that



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:

Then by property of FIX we have:

```
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
0)))
   =*> 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 =
$$\st -> (\x -> st p (x x)) (\x -> st p (x x))$$

How does it work? Let $FSTEP = |fAn \rightarrow ISE n ONE(MUL n (f (DEC n)))$

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)
```

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

88 of 88 9/23/21, 12:07 PM