Lambda Calculus

Your Favorite Language

Probably has lots of features:

- Assignment (`x = x + 1`)
- Booleans, integers, characters, strings, ...
- Conditionals
- Loops
- `return` , `break` , `continue`
- Functions
- Recursion
- References / pointers
- Objects and classes
- Inheritance
- ...

Which ones can we do without?

What is the **smallest universal language**?

name

def foo(...):

[

What is computable?

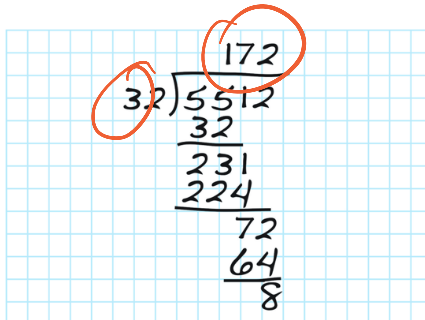
inputs, produce, output

"operations"

"simplification"

Before 1930s

Informal notion of an **effectively calculable** function:

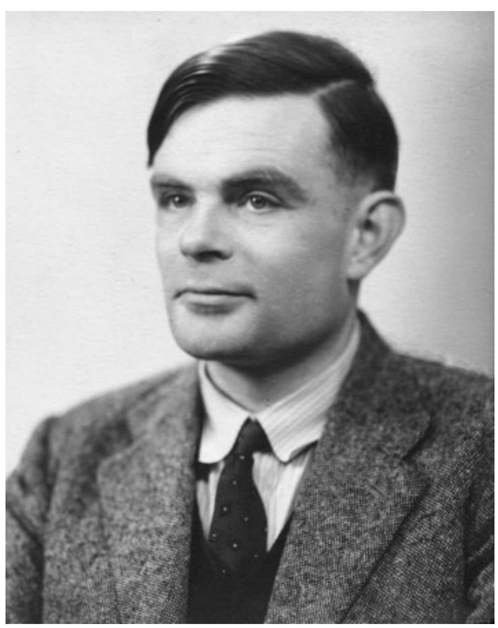


"doing stuff"

can be computed by a human with pen and paper, following an algorithm

1936: Formalization

What is the **smallest universal language**?



UK

Alan Turing

Alonzo Church

1930s

John McCarthy

LISP

1950s

SAIL

The Next 700 Languages

Peter Landin

> *Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.*

Peter Landin, 1966

The Lambda Calculus

Has one feature:

- Functions

No, *really*

- ~~Assignment (`x = x + 1`)~~
- ~~Booleans, integers, characters, strings, …~~
- ~~Conditionals~~
- ~~Loops~~
- ~~`return` , `break` , `continue`~~
- Functions
- ~~Recursion~~
- ~~References / pointers~~
- ~~Objects and classes~~
- ~~Inheritance~~
- ~~Reflection~~

$$(\text{function}(x)\{\, x \,\})(y) \rightarrow y$$

More precisely, *only thing* you can do is:

- **Define** a function
- **Call** a function

Describing a Programming Language

- *Syntax:* what do programs look like?
- *Semantics:* what do programs mean?
    - *Operational semantics*: how do programs execute step-by-step?

Syntax: What Programs Look Like

```
e ::= x  y, z, ...
    | (\x -> e)
    | (e e)
```

func arg

$E := x, y, z, \ldots$ vars

$|$ ~~function~~ λ$(x)\{\ E\ \}$ funcs

$|\ E(E)$ calls

Programs are **expressions** `e` (also called **λ-terms**) of one of three kinds:

"abstraction" = fun def

"application" = fun call

- **Variable**
    - `x`, `y`, `z`
- **Abstraction** (aka *nameless* function definition)
    - `(\x -> e)`
    - `x` is the *formal* parameter, `e` is the *body*
    - "for any `x` compute `e`"
- **Application** (aka function call)

- `(e1 e2)`
- `e1` is the *function*, `e2` is the *argument*
- in your favorite language: `e1(e2)`

(Here each of `e`, `e1`, `e2` can itself be a variable, abstraction, or application)

Examples

```
\x -> x                -- The "identity function" (id)
                       -- ("for any x compute x")

\x -> (\y -> y)        -- A function that returns (id)

\f -> (f (\x -> x))    -- A function that applies its argument to id
```

e ::= x, y, z, ...
| (\x -> e)
| (e e)
not a thing ~~| \e -> e~~

QUIZ

Which of the following terms are syntactically **incorrect**?

ie NOT valid LC express

A. `\(\x -> x) -> y`

$\underbrace{\texttt{(\textbackslash x -> x)}}_{\text{"var"}}$ "func that takes (\x->x) as input"

B. `\x -> x x`

C. `\x -> x (y x)`

D. A and C

E. all of the above

\x → (\y → ...)

Examples

```
\x -> x               -- The identity function
                      -- ("for any x compute x")

\x -> (\y -> y)       -- A function that returns the identity function

\f -> f (\x -> x)     -- A function that applies its argument
                      -- to the identity function
```

How do I define a function with two arguments?

- e.g. a function that takes `x` and `y` and returns `y`?

```
\x -> (\y -> y)      -- A function that returns the identity function
                     -- OR: a function that takes two arguments
                     -- and returns the second one!
```

How do I apply a function to two arguments?

- e.g. apply `\x -> (\y -> y)` to `apple` and `banana`?

(((\x → (\y → ...)) apple) banana)

(\ x y → y) apple banana

```
(((\x -> (\y -> y)) apple) banana) -- first apply to apple,
                                   -- then apply the result to banana
```

Syntactic Sugar

((x y) z)   x (y z)

instead of	**we write**
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`

```
\x y -> y       -- A function that that takes two arguments
                -- and returns the second one...

(\x y -> y) apple banana -- ... applied to two arguments
```

Semantics : What Programs Mean

How do I "run" / "execute" a λ-term?

Think of middle-school algebra:

```
-- Simplify expression:
(1 + 2) * ((3 * 8) - 2)
=
3       * ((3 * 8) - 2)
=
3       * ( 24     - 2)
=
3       * 22
=
66
```

$e \to e_1 \to e_2 \to \cdots \to e_k$

rewr

rew

rew

$(1+2) * (3-0)$

$= (1+2) * 3$

$= 3 * 3$

"redex" $= 9$

Execute = rewrite step-by-step

- Following simple *rules*
- until no more rules *apply*

$(\backslash x \to e_1)\; e_2 \Longrightarrow e_1[x := e_2]$

(\x -> x) apple

⇒ apple

(\x -> y) apple

⇒ y

1. apple

2. y

Rewrite Rules of Lambda Calculus

1. β-step (aka *function call*)
2. α-step (aka *renaming formals*)

e := x, y, z... ① "use/acc"

| (\x -> e) ② "define"

| (e e) ③

Where are variables "introduced"

But first we have to talk about **scope**

...“fuH” (\z -> stuff) ...“duH” z

z in scope

"The scope of a variable is the parts/region of the code where you can access that variable"

Semantics: Scope of a Variable

The part of a program where a **variable is visible**

In the expression `(\x -> e)`

- `x` is the newly introduced variable
- `e` **is the scope** of `x`
- any occurrence of `x` in `(\x -> e)` is **bound** (by the **binder** `\x`)

For example, `x` is bound in:

```
(\x -> x)
(\x -> (\y -> x))
```

x

(\x -> (y (\y -> x)))

An occurrence of `x` in `e` is **free** if it’s *not bound* by an enclosing abstraction

For example, `x` is free in:

```
(x y)                -- no binders at all!
(\y -> x y)          -- no \x binder
(\x -> (\y -> y)) x  -- x is outside the scope of the \x binder;
                     -- intuition: it's not "the same" x
```

QUIZ

1 2

In the expression `(\x -> x) x`, is `x` *bound* or *free*?

A. first occurrence is bound, second is bound x

B. first occurrence is bound, second is free ✓

C. first occurrence is free, second is bound x

D. first occurrence is free, second is free x

EXERCISE: Free Variables

An variable `x` is **free** in `e` if *there exists* a free occurrence of `x` in `e`

We can formally define the set of *all free variables* in a term like so:

```
FV(x)         = ???  {x}
FV(\x -> e)   = ???  FV(e) - {x}
FV(e1 e2)     = ???  FV(e1) ∪ FV(e2)
```

$FV(\backslash x \to x) = \emptyset$

$FV(\backslash x \to y) = \{y\}$

$FV((\backslash x \to x)\, x) = \{x\}$

$FV(x\ y) = \{x, y\}$

$FV(x) = \{x\}$

Closed Expressions

If `e` has *no free variables* it is said to be **closed**

- Closed expressions are also called **combinators**

Y - COMBINATOR "STARTUP INCUBATOR"

What is the shortest closed expression?

$(\backslash x \to x)$

Rewrite Rules of Lambda Calculus

1. β-step (aka *function call*)
2. α-step (aka *renaming formals*)

Semantics: Redex

$n_1 \oplus n_2$

A **redex** is a term of the form

`(\x -> e1) e2` $\Rightarrow_{\beta}$ $e_1 [x := e_2]$

A *function* `(\x -> e1)`

- `x` is the *parameter*
- `e1` is the *returned* expression

Applied to an argument `e2`

- `e2` is the *argument*

(\x -> x) apple

=b> apple

(\x -> (\x -> x)) apple

1 =b> \apple -> apple

2 =b> \apple -> x

3 =b> \x -> apple

4 =b> \x -> x

5 =b> WTF is going on!!!

Semantics: β-Reduction

A **redex** b-steps to another term ...

```
(\x -> e1) e2    =b>    e1[x := e2]
```

where `e1[x := e2]` means

“ `e1` with all *free* occurrences of `x` replaced with `e2` ”

Computation by *search-and-replace*:

- If you see an *abstraction* applied to an *argument*, take the *body* of the abstraction and replace all free occurrences of the *formal* by that *argument*
- We say that `(\x -> e1) e2` β-steps to `e1[x := e2]`

Redex Examples

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

Is this right? Ask Elsa (https://elsa.goto.ucsd.edu/index.html#?demo=permalink%2F1695925711_23.lc)

QUIZ

```
(\x -> (\y -> y)) apple
=b> ???
```

A. `apple`

B. `\y -> apple`

C. `\x -> apple`

D. `\y -> y`

E. `\x -> y`

QUIZ

((a b) c) d)

```
(\x -> y x y x) apple
=b> ???
```

A. `apple apple apple apple`

B. `y apple y apple`

C. `y y y y`

D. `apple`

QUIZ

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

Redex?
NO! lam-on-LEFT

A. `apple (\x -> x)`

((e1 e2) e3)

B. `apple (\apple -> apple)`

C. `apple (\x -> apple)`

D. `apple`

(\x -> x ((\x->x) banana)) apple

E. `\x -> x`

EXERCISE

What is a λ-term `fill_this_in` such that

```
fill_this_in apple
=b> banana
```

ELSA: https://elsa.goto.ucsd.edu/index.html

Click here to try this exercise (https://elsa.goto.ucsd.edu/index.html#?demo=permalink%2F1585434473_24432.lc)

A Tricky One

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

Is this right?

Something is Fishy

z

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

=a> (\x -> (\giraffe -> x)) y

=b> (\giraff -> y)

Is this right?

Problem: The *free* `y` in the argument has been **captured** by `\y` in *body*!

Solution: Ensure that *formals* in the body are **different from** *free-variables* of argument!

Capture-Avoiding Substitution

We have to fix our definition of β-reduction:

```
(\x -> e1) e2    =b>    e1[x := e2]
```

where `e1[x := e2]` means "~~`e1` with all *free* occurrences of `x` replaced with `e2`~~"

- `e1` with all *free* occurrences of `x` replaced with `e2`
- **as long as** no free variables of `e2` get captured

Formally:

```
cat [cat := horse]  →  horse
x[x := e]                = e

cat [dog := horse]  →  cat
y[x := e]                = y                  -- as x /= y

(e1 e2)[x := e]          = (e1[x := e]) (e2[x := e])

(\cat → cat) [cat := dog]
(\x -> e1)[x := e]       = \x -> e1           -- Q: Why `e1` unchanged?

(\y → x) [x := y]
(\y -> e1)[x := e]
  | not (y in FV(e)) = \y -> e1[x := e]
```

Oops, but what to do if `y` is in the *free-variables* of `e`?

- i.e. if `\y -> ...` may *capture* those free variables?

Rewrite Rules of Lambda Calculus

1. β-step (aka *function call*)
2. α-step (aka *renaming formals*)

Semantics: α-Renaming

z

x

(\y -> x)

```
\x -> e   =a>   \y -> e[x := y]
  where not (y in FV(e))
```

- We rename a formal parameter `x` to `y`
- By replace all occurrences of `x` in the body with `y`
- We say that `\x -> e` α-steps to `\y -> e[x := y]`

Example:

```
\x -> x   =a>   \y -> y   =a>    \z -> z
```

All these expressions are **α-equivalent**

What's wrong with these?

```
-- (A)
\f -> f x    =a>   \x -> x x
```

```
-- (B)
(\x -> \y -> y) y   =a>   (\x -> \z -> z) z
```

Tricky Example Revisited

```
(\x -> (\y -> x)) y
                                -- rename 'y' to 'z' to avoid capture
=a> (\x -> (\z -> x)) y
                                -- now do b-step without capture!
=b> \z -> y
```

To avoid getting confused,

- you can **always rename** formals,
- so different **variables** have different **names**!

Normal Forms

Recall **redex** is a λ-term of the form

```
(\x -> e1) e2
```

A λ-term is in **normal form** if it *contains no redexes.*

QUIZ

Which of the following term are **not** in *normal form* ?

A. `x`

B. `x y`

x **C.** `(\x -> x) y` → y

D. `x (\y -> y)`

E. C and D

Semantics: Evaluation

A λ-term `e` **evaluates to** `e'` if

1. There is a sequence of steps

```
e =?> e_1 =?> ... =?> e_N =?> e'
```

where each `=?>` is either `=a>` or `=b>` and `N >= 0`

2. `e'` is in *normal form*

Examples of Evaluation

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

formal body arg

```
(\f -> f (\x -> x)) (\x -> x)
  =?> ???
```

(\x→x) (\x→x)

=b> (\x→x)

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

Elsa shortcuts

Named λ-terms:

```
let ID = \x -> x  -- abbreviation for \x -> x
```

To substitute name with its definition, use a `=d>` step:

```
ID apple
  =d> (\x -> x) apple    -- expand definition
  =b> apple              -- beta-reduce
```

Evaluation:

- `e1 =*> e2` : `e1` reduces to `e2` in 0 or more steps
  - where each step is `=a>` , `=b>` , or `=d>`
- `e1 =~> e2` : `e1` evaluates to `e2` and `e2` is **in normal form**

EXERCISE

Fill in the definitions of `FIRST` , `SECOND` and `THIRD` such that you get the following behavior in `elsa`

```
let FIRST  = fill_this_in
let SECOND = fill_this_in
let THIRD  = fill_this_in
```

$(\backslash x_1 \rightarrow (\backslash x_2 \rightarrow (\backslash x_3 \rightarrow x_1)))$

```
eval ex1 :
  (((FIRST apple) banana) orange)
  =*> apple
```

$(\backslash x_1 \rightarrow (\backslash x_2 \rightarrow (\backslash x_3 \rightarrow x_2)))$

```
eval ex2 :
  (((SECOND apple) banana) orange)
  =*> banana
```

$(\backslash x_1 \rightarrow (\backslash x_2 \rightarrow (\backslash x_3 \rightarrow x_3)))$

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

ELSA: https://goto.ucsd.edu/elsa/index.html

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434130_24421.lc)

Non-Terminating Evaluation

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

Some programs loop back to themselves…

… and *never* reduce to a normal form!

This combinator is called Ω

What if we pass Ω as an argument to another function?

```
let OMEGA = (\x -> x x) (\x -> x x)
```

```
(\x -> (\y -> y)) OMEGA
```

Does this reduce to a normal form? Try it at home!

Programming in λ-calculus

Real languages have lots of features

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

Lets see how to *encode* all of these features with the λ-calculus.

ITE = \ b x y -> (b x y)

2 values

"true" "false"

logical operators

- or, and, not, ...
- ite b e1 e2

"if b is true" -> x

"else" -> y

TRUE x y ≐ x

TRUE x ≐ \y -> x

TRUE = \x -> \y -> x

λ-calculus: Booleans

TRUE ≡ (\x -> (\y -> x))

FALSE ≡ (\x -> (\y -> y))

TRUE x y = x

FALSE x y = y

How can we encode Boolean values (`TRUE` and `FALSE`) as functions?

Foo x = e

Foo = \x -> e

Well, what do we **do** with a Boolean `b` ?

make a choice

Make a *binary choice*

- **if** b **then** e1 **else** e2

Booleans: API

We need to define three functions

```
let TRUE  = ???
let FALSE = ???
let ITE   = \b x y -> ???  -- if b then x else y
```

such that

```
ITE TRUE apple banana =~> apple
ITE FALSE apple banana =~> banana
```

(Here, `let NAME = e` means `NAME` is an *abbreviation* for `e`)

Booleans: Implementation

```
let TRUE  = \x y -> x         -- Returns its first argument
let FALSE = \x y -> y         -- Returns its second argument
let ITE   = \b x y -> b x y   -- Applies condition to branches
                              -- (redundant, but improves readability)
```

Example: Branches step-by-step

```
eval ite_true:
  ITE TRUE e1 e2
  =d> (\b x y -> b    x  y) TRUE e1 e2    -- expand def ITE
  =b>   (\x y -> TRUE x  y)      e1 e2    -- beta-step
  =b>     (\y -> TRUE e1 y)         e2    -- beta-step
  =b>            TRUE e1 e2               -- expand def TRUE
  =d>     (\x y -> x) e1 e2               -- beta-step
  =b>       (\y -> e1)   e2               -- beta-step
  =b> e1
```

Example: Branches step-by-step

Now you try it!

Can you fill in the blanks to make it happen? (https://elsa.goto.ucsd.edu/index.html#?demo=ite.lc)

```
eval ite_false:
  ITE FALSE e1 e2

  -- fill the steps in!

  =b> e2
```

EXERCISE: Boolean Operators

ELSA: https://goto.ucsd.edu/elsa/index.html Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585435168_24442.lc)

Now that we have `ITE` it's easy to define other Boolean operators:

```
let NOT = \b     -> ???
let OR  = \b1 b2 -> ???
let AND = \b1 b2 -> ???
```

When you are done, you should get the following behavior:

```
eval ex_not_t:
  NOT TRUE =*> FALSE

eval ex_not_f:
  NOT FALSE =*> TRUE

eval ex_or_ff:
  OR FALSE FALSE =*> FALSE

eval ex_or_ft:
  OR FALSE TRUE =*> TRUE

eval ex_or_ft:
  OR TRUE FALSE =*> TRUE

eval ex_or_tt:
  OR TRUE TRUE =*> TRUE

eval ex_and_ff:
  AND FALSE FALSE =*> FALSE

eval ex_and_ft:
  AND FALSE TRUE =*> FALSE

eval ex_and_ft:
  AND TRUE FALSE =*> FALSE

eval ex_and_tt:
  AND TRUE TRUE =*> TRUE
```

Programming in λ-calculus

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

collection of things

↳ "get" one thing from collecti

↳ "set" / "create" collection

FST (MkPair t_1 t_2) $= t_1$

SND (MkPair t_1 t_2) $= t_2$

MkPair = \ t_1 t_2 -> (\ b -> ITE b t_1 t_2)

FST = \p -> p TRUE

SND = \p -> p FALSE

λ-calculus: Records

Let's start with records with *two* fields (aka **pairs**)

What do we *do* with a pair?

1. **Pack two** items into a pair, then
2. **Get first** item, or
3. **Get second** item.

Pairs : API

We need to define three functions

```
let PAIR = \x y -> ???     -- Make a pair with elements x and y
                           -- { fst : x, snd : y }
let FST  = \p -> ???       -- Return first element
                           -- p.fst
let SND  = \p -> ???       -- Return second element
                           -- p.snd
```

such that

```
eval ex_fst:
  FST (PAIR apple banana) =*> apple

eval ex_snd:
  SND (PAIR apple banana) =*> banana
```

Pairs: Implementation

A pair of `x` and `y` is just something that lets you pick between `x` and `y`! (i.e. a function that takes a boolean and returns either `x` or `y`)

```
let PAIR = \x y -> (\b -> ITE b x y)
let FST  = \p -> p TRUE   -- call w/ TRUE, get first value
let SND  = \p -> p FALSE  -- call w/ FALSE, get second value
```

EXERCISE: Triples

How can we implement a record that contains **three** values?

ELSA: https://goto.ucsd.edu/elsa/index.html

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
```

Programming in λ-calculus

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

create new bools from old (eg NOT, AND) / "construct" / "build"

do "choice" (ITE) / "destruct" / "use"

3 5 7

||| ||||| |||||||

zero + "succ"/"next"

"5" = do an op 5 times $\backslash f\ x \to f(f(f(f(f\,x))))$

"7" = do an op 7 times $\backslash f\ x \to f(f(f(f(f(f(f\,x))))))$

"0" = do an op 0 times $\backslash f\ x \to x$

"1" = do an op 1 times $\backslash f\ x \to f\ x$

$$\text{"}n\text{"} = \backslash f\ x \to \underbrace{f(\ldots f(f(f\,x)))}_{n\text{-times}}$$

$$(n\ f\ x) \equiv f^n(x)$$

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:

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

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

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

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)))))
...
```

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`   ONE !
- **D**: `let ZERO = \x -> 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 -> ???)
```

$$INC\ n = \backslash f\ x \to \underbrace{f(\overbrace{\cdots f(f(f(x)))}^{n \text{ times}})}_{n+1 \text{ times}}$$

$$(n\ f\ x)$$

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

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

$$n\ f\ x = f^{n}(x)$$

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
```

$n_1 + n_2$

```
eval add_zero_zero:
  ADD ZERO ZERO =~> ZERO
```

$(n_2 + 1) \underbrace{+1 +1 + \dots}_{n_1}$

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

$\underbrace{\text{INC} \dots (\text{INC} (\text{INC}}_{n_1} (\text{INC } n_2)))$

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

$n_1 \ \text{INC} \ n_2$

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

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

$\underbrace{(n_2 + \dots (n_2 + (n_2 +}_{n_1 \text{ times}} \text{ZERO})))$

```
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`

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

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

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

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

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)`

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

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
```

IS_ZERO ZERO → TRUE

IS_ZERO ONE → FALSE

IS_ZERO TWO → FALSE

Programming in λ-calculus

- ✓ **Booleans** [done]

Haskell

- **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

HEAD

```
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

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

λ-calculus: Lists

"fst" "mkPair"

HEAD (CONS h t)

=*> h

"snd

TAIL (CONS h t)

=*> t

```
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

Z = \f x → x
ONE = \f x → f x
TWO = \f x → f (f x)

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 = ???
```

= \n l → "call tail n times, then get head"

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

n l

```
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 = \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**?

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

SUM

n f x

```
res = x
i = 0
while i < n :
    res = f(res)
    i += 1

snd (res)
```

def f(p):

res₀ → res₁ → res₂ → ... → resₙ (each step f)

$\boxed{2+1+0}$

n+(n-1)+...+1+0

(1, 0)
(2, 1+0)
(3, 2+1+0)
(n, n+...+1+0)

(i, total)
↓
(i+1, i+total)

\p → Pair ("i+1") ("i + total")

f_0 = (\p → let i = FST p
let total = SND p
PAIR (ADD ONE i) (ADD i tot))

x_0 = PAIR ONE ZERO

SUM = \n → SND (n f_0 x_0)

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:

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

Instead of

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

Lets try

"bob"

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

"bob"

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
```

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)
```

FIX STEP ≡ STEP (FIX STEP)

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

$x \longrightarrow f(x) \longrightarrow f(f(x)) \cdots \quad f^n(x) \overset{?}{=} f^{n+1}(x)$

Once we have it, we can define:

SUM →> STEP SUM

```
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 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))
```

Fix STEP $\rightarrow^*$ STEP (Fix STEP)

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

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/fa23/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/).