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**?
What is computable?

Before 1930s

Informal notion of an **effectively calculable** function:
can be computed by a human with pen and paper, following an algorithm

1936: Formalization

What is the smallest universal language?
Alan Turing
The Next 700 Languages
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 (\( x \leftarrow x + 1 \) )
- Booleans, integers, characters, strings, ...
- Conditionals
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

\[ e ::= x, y, z, \text{apple}, \ldots \text{ variables} \]

- \[ (\lambda x \rightarrow e) \text{ -- function that takes a parameter 'x' and returns 'e'} \]
- \[ (e_1 e_2) \text{ -- call (function) 'e1' with argument 'e2'} \]

Programs are expressions \( e \) (also called \( \lambda \)-terms) of one of three kinds:

- **Variable**
  - \( x, y, z \)

- **Abstraction** (aka nameless function definition)
  - \( (\lambda x \rightarrow e) \)
  - \( x \) is the formal parameter, \( e \) is the body
  - “for any \( x \) compute \( e \)”

- **Application** (aka function call)
  - \( e_1(e_2) \)
○ (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
\( \text{function}\ (x) \equiv \text{return } x^3 \)

\( (\text{x} \to \text{x}) \)  -- The identity function \((\text{id})\) that returns its input

\( \text{function}\ (x)\{\text{return } \text{function}\ (y) \equiv \text{return } y^3\}\); 

\( (\text{x} \to (\text{y} \to \text{y})) \)  -- A function that returns \((\text{id})\)

\( (\text{f} \to (\text{f} \to \text{x})) \)  -- A function that applies its argument to \(\text{id}\)

fun (f) \equiv \text{return } f\ (\text{function}\ (x) \equiv \text{return } x^3)\)

---

**QUIZ**

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

\[ e = \alpha 1 (\lambda x \to e ) \mid (e_1, e_2) \]
Examples
(\x -> x) -- The identity function (id) that returns its input

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

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

How do I define a function with two arguments?

  • e.g. a function that takes x and y and returns y?
How do I apply a function to two arguments?

- e.g. apply \((\ x \rightarrow \ y\ )\ )\ to \ apple\ and\ banana?
(((\x -> (\y -> y)) apple) banana) -- first apply to apple,
-- then apply the result to banana
### Syntactic Sugar

<table>
<thead>
<tr>
<th>instead of</th>
<th>we write</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x \rightarrow (\ y \rightarrow (\ z \rightarrow e)))</td>
<td>(x \rightarrow \ y \rightarrow \ z \rightarrow e)</td>
</tr>
<tr>
<td>(x \rightarrow \ y \rightarrow \ z \rightarrow e)</td>
<td>(x \ y \ z \rightarrow e)</td>
</tr>
<tr>
<td>(((e1 e2) e3) e4)</td>
<td>e1 e2 e3 e4</td>
</tr>
</tbody>
</table>

\[ x \ y \rightarrow y \] -- A function that takes two arguments  
-- and returns the second one...

\((\ x \ y \rightarrow y\) apple banana -- ... applied to two arguments
Semantics: What Programs Mean

How do I “run” / “execute” a $\lambda$-term?

Think of middle-school algebra:

\[
(1 + 2) * ((3 * 8) - 2) =\]
\[
3 * ((3 * 8) - 2) =\]
\[
3 * (24 - 2) =\]
\[
3 * 22 =\]
\[
66
\]
**Execute** = rewrite step-by-step

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

---

**Rewrite Rules of Lambda Calculus**

1. **β**-step (aka *function call*)
2. **α**-step (aka *renaming formals*)
But first we have to talk about **scope**

**Semantics: Scope of a Variable**

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

In the expression \( (\lambda x \rightarrow e) \)

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

For example, \( x \) is bound in:
An occurrence of \( x \) in \( e \) is free if it's not bound by an enclosing abstraction.

For example, \( x \) is free in:

\[
(x \ y) \quad - \quad \text{no binders at all!}
\]

\[
(y \rightarrow (x \ y)) \quad - \quad \text{no} \ x \ \text{binder}
\]

\[
((x \rightarrow (y \rightarrow y)) \ x) \quad - \quad \text{x is outside the scope of the} \ x \ \text{binder;}
\]

\[
\text{-- intuition: it's not "the same" x}
\]
**QUIZ**

Is $x$ *bound* or *free* in the expression $((\lambda x \to x) \ x)$?

A. first occurrence is bound, second is bound

B. first occurrence is bound, second is free

C. first occurrence is free, second is bound

D. first occurrence is free, second is free
**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:

- $FV(x) = ???$
- $FV(\lambda x \rightarrow e) = ???$
- $FV(e_1 e_2) = ???$
Closed Expressions

If \( e \) has no free variables it is said to be closed

- Closed expressions are also called combinators

What is the shortest closed expression?
Rewrite Rules of Lambda Calculus

1. $\beta$-step (aka function call)
2. $\alpha$-step (aka renaming formals)

Semantics: Redex
A **redex** is a term of the form

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

A **function** `\(\x \rightarrow e1\)`

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

*Applied to* an argument `e2`

- `e2` is the *argument*
Semantics: $\beta$-Reduction

A **redex** $\beta$-steps to another term ...

$$(\lambda x \to e_1)\, e_2 =_b e_1[x := e_2]$$

where $e_1[x := e_2]$ means

“$e_1$ with all *free* occurrences of $x$ replaced with $e_2$”

Computation by *search-and-replace*:

If you see an *abstraction* applied to an *argument*,

- In the *body* of the abstraction
- Replace all *free* occurrences of the *formal* by that *argument*

We say that $(\lambda x \to e_1)\, e_2$ $\beta$-steps to $e_1[x := e_2]$
Redex Examples

((\x -> x) apple)

=\b> apple

Is this right? Ask Elsa (https://goto.ucsd.edu/elsa/index.html)
QUIZ

[((\x -> (\y -> y)) apple)

=b> ???

A. apple
B. \y -> apple
C. \x -> apple
D. \y -> y
E. \x -> y
QUIZ

\( (\lambda x \to (((y x) y) x)) \text{ apple} \)

= b> ???

A. \(((\text{apple } \text{apple}) \text{ apple}) \text{ apple})\)

B. \(((y \text{ apple}) y) \text{ apple})\)

C. \(((y y) y) y\)

D. apple
QUIZ

$$((\lambda x \rightarrow (x (\lambda x \rightarrow x))) \text{ apple})$$

= b> ???

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

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

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

E. (\x \to x)

**EXERCISE**

What is a \(\lambda\)-term \texttt{fill\_this\_in} such that

\texttt{fill\_this\_in apple} \\
\(=b\) \texttt{banana}
ELSA: https://goto.ucsd.edu/elsa/index.html

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

A Tricky One

$$\left(\left(\lambda x \rightarrow \left(\lambda y \rightarrow x\right)\right) y\right)$$

=\(\lambda y \rightarrow y\)
Is this right?

Something is Fishy

\((\lambda x \rightarrow (\lambda y \rightarrow x)) \ y\)

=b> (\lambda y \rightarrow y)

Is this right?
Problem: The free $y$ in the argument has been captured by $\backslash y$ 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 $\beta$-reduction:

$$(\lambda x \to e_1) e_2 \Rightarrow e_1[x := e_2]$$
where $e_1[x := e_2]$ means “$e_1$ with all free occurrences of $x$ replaced with $e_2$”

- $e_1$ with all free occurrences of $x$ replaced with $e_2$
- as long as no free variables of $e_2$ get captured

Formally:

$x[x := e] = e$

$y[x := e] = y$ \quad -- \text{as } x \neq y$

$(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$

$(\lambda x \to e_1)[x := e] = (\lambda x \to e_1)$ \quad -- Q: Why leave `e1` unchange
d

$(\lambda y \to e_1)[x := e]$

| not (y in FV(e)) = \lambda y \to e_1[x := e]$

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

- i.e. if \( \lambda y \to \ldots \) may capture those free variables?
Rewrite Rules of Lambda Calculus

1. $\beta$-step (aka function call)
2. $\alpha$-step (aka renaming formals)
Semantics: $\alpha$-Renaming

\[ \lambda x \rightarrow e \ =a> \ \lambda y \rightarrow e[x := y] \]
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 $\lambda x \rightarrow e \ \alpha$-steps to $\lambda y \rightarrow e[x := y]$

Example:

$(\lambda x \rightarrow x) \ =a> \ (\lambda y \rightarrow y) \ =a> \ (\lambda z \rightarrow z)$

All these expressions are $\alpha$-equivalent
What's wrong with these?

-- (A)
\( (\lambda f \to (f \ x)) \ =_{a>} \ (\lambda x \to (x \ x)) \)

-- (B)
\( ((\lambda x \to (\lambda y \to y)) \ y) \ =_{a>} \ ((\lambda x \to (\lambda z \to z)) \ z) \)
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 \textbf{redex} is a $\lambda$-term of the form

$$((\backslash x \rightarrow e_1) e_2)$$

A $\lambda$-term is in \textbf{normal form} if it \textit{contains no redexes}.
QUIZ

Which of the following term are not in normal form?

A. x
B. (x y)
C. ((\x -> x) y)
D. (x (\y -> y))
E. C and D
Semantics: Evaluation

A $\lambda$-term $e$ evaluates to $e'$ if

1. There is a sequence of steps

$$e = ? > e_1 = ? > \ldots = ? > e_N = ? > e'$$

where each $=? >$ is either $=a>$ or $=b>$ and $N \geq 0$

2. $e'$ is in normal form

Examples of Evaluation

$$((\lambda x \rightarrow x)\;\text{apple})$$

$=b > \text{apple}$
\(\left(\lambda f \rightarrow f \left(\lambda x \rightarrow x\right)\right) \left(\lambda x \rightarrow x\right)\)

=???

\(\left(\lambda x \rightarrow x \ x\right) \left(\lambda x \rightarrow x\right)\)

=???

---

**Elsa shortcuts**

Named \(\lambda\)-terms:

```latex
let ID = (\x \rightarrow x) \quad -- \textit{abbreviation for} \ (\x \rightarrow x)
```
To substitute name with its definition, use a =d> step:

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

Evaluation:

- $e_1 =*> e_2$: $e_1$ reduces to $e_2$ in 0 or more steps
  - where each step is =a>, =b>, or =d>
- $e_1 =~> e_2$: $e_1$ evaluates to $e_2$ and $e_2$ 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
```

```
eval ex1 :
    FIRST apple banana orange
    => apple
```

```
eval ex2 :
    SECOND apple banana orange
    => banana
```

```
eval ex3 :
    THIRD apple banana orange
    => orange
```

ELSA: https://goto.ucsd.edu/elsa/index.html
Non-Terminating Evaluation

\((\lambda x . (x x)) (\lambda x . (x x))\)

= \(\infty\) \(\infty\)

Some programs loop back to themselves ... never reduce to a normal form!

This combinator is called \(\Omega\)

What if we pass \(\Omega\) as an argument to another function?

\[
\text{let } \text{OMEGA} = (\lambda x . (x x)) (\lambda x . (x x))
\]

\[
(\lambda x . (\lambda y . y)) \text{OMEGA}
\]
Does this reduce to a normal form? Try it at home!

Programming in $\lambda$-calculus

Real languages have lots of features

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

Lets see how to encode all of these features with the $\lambda$-calculus.
**Syntactic Sugar**

<table>
<thead>
<tr>
<th>instead of</th>
<th>we write</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\x -&gt; (\y -&gt; (\z -&gt; e))</code></td>
<td><code>\x -&gt; \y -&gt; \z -&gt; e</code></td>
</tr>
<tr>
<td><code>\x -&gt; \y -&gt; \z -&gt; e</code></td>
<td><code>\x \y \z -&gt; e</code></td>
</tr>
<tr>
<td><code>(((e1 e2) e3) e4)</code></td>
<td><code>e1 e2 e3 e4</code></td>
</tr>
</tbody>
</table>
\( \lambda \) \text{-calculus: Booleans

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

Well, what do we do with a Boolean b?
Make a *binary choice*

- \( \text{if } b \text{ then } e_1 \text{ else } e_2 \)

**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)
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
=\( b \ x \ y \rightarrow b \ x \ y \) TRUE e1 e2 \quad \text{-- expand def ITE}
=\( \ x \ y \rightarrow \text{TRUE} \ x \ y \) \ e1 \ e2 \quad \text{-- beta-step}
=\( \ y \rightarrow \text{TRUE} \ e1 \ y \) \ e2 \quad \text{-- beta-step}
=\( \ y \rightarrow \text{TRUE} \ e1 \ e2 \) \quad \text{-- expand def TRUE}
=\( \ x \ y \rightarrow x \) \ e1 \ e2 \quad \text{-- beta-step}
=\( \ y \rightarrow e1 \) \ e2 \quad \text{-- beta-step}
=b> e1

Example: Branches step-by-step

Now you try it!

Can you fill in the blanks to make it happen? (http://goto.ucsd.edu:8095/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
Now that we have ITE it’s easy to define other Boolean operators:

```latex
let NOT = \b \rightarrow ???
let OR = \b1 \ b2 \rightarrow ???
let AND = \b1 \ b2 \rightarrow ???
```

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 $\lambda$-calculus

- **Booleans** [done]
- **Records** (structs, tuples)
- Numbers
- Lists

\[
\text{ITE } b \text{ } X_1 \text{ } X_2
\]

\[
\begin{array}{c|c|c}
\text{TRUE} & \text{FALSE} \\
\end{array}
\]
Functions [we got those]
Recursion

**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 **PACK**
2. **Get first** item, or **FST**
3. **Get second** item. **SND**
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 $y$!

```latex
let PAIR = \[x \quad y \rightarrow (\{b \rightarrow \text{ITE } b \quad x \quad y\})\]
```

i.e. $\text{PAIR } x \quad y$ is a function that

- takes a boolean and returns either $x$ or $y$

We can now implement $\text{FST}$ and $\text{SND}$ by “calling” the pair with $\text{TRUE}$ or $\text{FALSE}$

```latex
let FST = \[p \rightarrow p \quad \text{TRUE} \quad \quad \text{-- call w/ TRUE, get first value}\]
let SND = \[p \rightarrow p \quad \text{FALSE} \quad \quad \text{-- call w/ FALSE, get second value}\]
```
**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)

```
TRIPLE = \( v_1, v_2, v_3 \mapsto \PAIR v_1 (\PAIR v_2 v_3) \)
FST3 = \( \tup \mapsto \FST \tup \)
SND3 = \( \tup \mapsto \FST (\SND \tup) \)
THD3 = \( \tup \mapsto \SND (\SND \tup) \)
```
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 $\lambda$-calculus

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

```
3 = \ x . f (f (f x))
5 = f f f f f f
"3" = \ f . x . f (f (f x))
"5" = \ f . x . f (f (f (f (f x))))
```
\[\textbf{\lambda-calculus: Numbers}\]

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

What do we do with natural numbers?

- Count: 0, \texttt{inc}
- Arithmetic: \texttt{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
- ...


---

The image contains a page from a document discussing the concept of natural numbers, focusing on defining a family of numerals and arithmetic functions. It emphasizes the need for these definitions to respect all regular laws of arithmetic.
Natural Numbers: Implementation

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

```plaintext
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)))))
...
```

```
"n" = \f x -> f (\ldots (f x))
    \quad \text{"n" times}
```

```
(n \, f \, x) = f (f (f (f (\ldots (f x)))))
    \quad \text{"n" times}
```
QUIZ: Church Numerals

Which of these is a valid encoding of \(\text{ZERO} \) ?

- **A**: \( \text{let } \text{ZERO} = \lambda x \rightarrow x \)
  - \( = \text{FALSE} \)
- **B**: \( \text{let } \text{ZERO} = \lambda f \rightarrow f \)
- **C**: \( \text{let } \text{ZERO} = \lambda f \rightarrow f \ x \)
- **D**: \( \text{let } \text{ZERO} = \lambda x \rightarrow x \)
  - \( \times \)
- **E**: None of the above

Does this function look familiar?
\[ \text{INCREMENT} \quad \uparrow \downarrow \]

\begin{align*}
\text{INC ZERO} & \Rightarrow \text{ONE} \\
\text{INC ONE} & \Rightarrow \text{TWO} \\
\end{align*}

\[
\text{let INC} = \lambda n \rightarrow (\lambda f \ x \rightarrow f (n f x))
\]

\[
\text{let INC} = \lambda n \rightarrow (\lambda f \ x \rightarrow f(n f x))
\]

\[\lambda\text{-calculus: Increment} \quad \text{let INC } \quad = \lambda n \rightarrow (\lambda f \ x \rightarrow f(n f x))
\]

-- Call `f` on `x` one more time than `n` does

let INC = \lambda n \rightarrow (\lambda f \ x \rightarrow f(n f x))

= \lambda n \rightarrow (\lambda f \ x \rightarrow f(n f x))

= \lambda n \rightarrow (\lambda f \ x \rightarrow 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 \texttt{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)

\[
\begin{align*}
\text{ADD } n \ m &= \lambda f \ x \rightarrow n \ f \ (m \ f \ x) \\
\text{ADD } n \ m &= \lambda f \ x \rightarrow n \ \text{INC} \ m \\
&= \lambda f \ x \rightarrow m \ \text{INC} \ n
\end{align*}
\]
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_one:
  ADD ONE ONE ~> TWO

eval add_two_zero:
  ADD TWO ZERO ~> TWO
QUIZ

How shall we implement ADD?

A. let ADD = \( \lambda m \rightarrow n \ INC \ m \)

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

C. let ADD = \( \lambda n m \rightarrow n \ m \ INC \)

D. let ADD = \( \lambda n m \rightarrow n \ (m \ INC) \)

E. let ADD = \( \lambda n m \rightarrow n \ (INC \ m) \)

\[ (n \ INC \ (m \ INC) \ (m \ INC \ n)) \]

\[ ((N \ INC) \ m) \ vs \ (n \ (INC \ m)) \]

\[ \text{call f" m times"} \]

\[ \text{INC (INC (INC (INC ... (INC m)))}) \]

v m times
\(\lambda\)-calculus: Addition

-- Call `f` on `x` exactly `n + m` times

$$\text{let } \text{ADD} = \lambda n \ m . n \ \text{INC} \ m$$

Example:

```plaintext
eval add_one_zero :
  ADD \ ONE \ ZERO
  =~> \ ONE
```
**QUIZ**

How shall we implement \( \textsc{MULT} \)?

A. `let \textsc{MULT} = \lambda m \rightarrow n \ \textsc{ADD} \ m`

B. `let \textsc{MULT} = \lambda m \rightarrow n \ (\textsc{ADD} \ n) \ \textsc{ZERO}`

C. `let \textsc{MULT} = \lambda m \rightarrow m \ (\textsc{ADD} \ n) \ \textsc{ZERO}`

D. `let \textsc{MULT} = \lambda m \rightarrow n \ (\textsc{ADD} \ m) \ \textsc{ZERO}`

E. `let \textsc{MULT} = \lambda m \rightarrow (n \ \textsc{ADD} \ m) \ \textsc{ZERO}`
\[\lambda\text{-calculus: Multiplication}\]

\[\text{-- Call } f \text{ on } x \text{ exactly } n \times m \text{ times}\]

let \( \text{MULT} = \lambda n \ m \rightarrow n \ (\text{ADD} \ m) \ \text{ZERO} \)

Example:

\[
\text{eval two\_times\_three :}
\text{MULT TWO ONE}
\rightarrow TWO
\]
Programming in λ-calculus

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

\[
\begin{align*}
\text{ISZ ZERO} & = \text{TRUE} \\
\text{ISZ ONE} & = \text{FALSE} \\
\text{ISZ TWO} & = \text{FALSE}
\end{align*}
\]
**λ-calculus: Lists**

Let's 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

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

\textbf{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

- \( \text{GetNth } n \ l \) returns the \( n \)-th element of the list \( l \)

Assume that \( l \) has \( n \) or more elements

\[
\text{let GetNth = ???}
\]

\[
\text{eval nth1 : GetNth ZERO (CONS apple (CONS banana (CONS cantaloupe NIL)))}
\]
\[
\Rightarrow \text{apple}
\]

\[
\text{eval nth1 : GetNth ONE (CONS apple (CONS banana (CONS cantaloupe NIL)))}
\]
\[
\Rightarrow \text{banana}
\]

\[
\text{eval nth2 : GetNth TWO (CONS apple (CONS banana (CONS cantaloupe NIL)))}
\]
\[
\Rightarrow \text{cantaloupe}
\]

Click here to try this in elsa (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1586466816_52273.lc)
I want to write a function that sums up natural numbers up to $n$:

\[
\text{let } \text{SUM } = \lambda n \rightarrow \ldots\ldots 0 + 1 + 2 + \ldots + n
\]

such that we get the following behavior:

- $\text{eval exSum0: SUM ZERO } \Rightarrow \text{ ZERO}$
- $\text{eval exSum1: SUM ONE } \Rightarrow \text{ ONE}$
- $\text{eval exSum2: SUM TWO } \Rightarrow \text{ THREE}$
- $\text{eval exSum3: SUM THREE } \Rightarrow \text{ SIX}$

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)

**QUIZ**

You can write $\text{SUM}$ using numerals but it's *tedious*.

Is this a correct implementation of $\text{SUM}$?

```latex
let \text{SUM} = \lambda n \rightarrow \text{ITE} (\text{ISZ} n)
\hspace{1cm} \text{ZERO}
\hspace{1cm} \text{(ADD n (SUM (DEC n)))}
```

A. Yes

B. No **Cheating**!
B. No

No!

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

```
\text{SUM} = \lambda n \to \text{ITE} (\text{ISZ} n) \text{ZERO} \text{ADD} n (\text{SUM} (\text{DEC} n)) \text{-- But SUM is not yet defined!}
```

Recursion:
• Inside this function
• Want to call the same function on \( \text{DEC } n \)

Looks like we can’t do recursion!

• Requires being able to refer to functions by name,
• But \( \lambda \)-calculus functions are anonymous.

Right?
\(\lambda\)-calculus: Recursion

Think again!

Recursion:

Instead of

- Inside this function I want to call the same function on \(\text{DEC} \ n\)

Lets try

- Inside this function I want to call some function \(\text{rec}\) on \(\text{DEC} \ n\)
- And BTW, I want \(\text{rec}\) to be the same function

**Step 1:** Pass in the function to call “recursively”

```lambda
let STEP =
\rec -> \n -> \text{ITE} (\text{ISZ} \ n)
      \text{ZERO}
      (\text{ADD} \ n (\ rec (\text{DEC} \ n))) -- Call some rec
```
**Step 2:** Do some magic to \texttt{STEP}, so \texttt{rec} is itself

\[
\lambda n \rightarrow \text{ITE} \left( \text{ISZ} \ n \right) \ \text{ZERO} \ (\text{ADD} \ n \ (\text{rec} \ (\text{DEC} \ n)))
\]

That is, obtain a term MAGIC such that

\[
\text{MAGIC} \ = \leftrightarrow \ \text{STEP} \ \text{MAGIC}
\]
Wanted: a $\lambda$-term $\text{FIX}$ such that

- $\text{FIX} \text{ STEP}$ calls $\text{STEP}$ with $\text{FIX} \text{ STEP}$ as the first argument:

$$(\text{FIX} \text{ STEP}) = \Rightarrow \text{STEP} (\text{FIX} \text{ STEP})$$

(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 \text{SUM} = \text{FIX} \text{ STEP}
```

Then by property of $\text{FIX}$ we have:

$\text{SUM} = \Rightarrow \text{FIX} \text{ STEP} = \Rightarrow \text{STEP} (\text{FIX} \text{ STEP}) = \Rightarrow \text{STEP} \text{ SUM}$

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?

\((\lambda x \rightarrow x \ x) \ (\lambda x \rightarrow x \ x)\)

= \(b > (\lambda x \rightarrow x \ x) \ (\lambda x \rightarrow x \ x)\)

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

\[\text{let } \text{FAC-s} = \text{rec } n \rightarrow \text{ITE} (\text{ist } n) \text{ ONE} (\text{MUL } n (\text{rec } (\text{DEC } n)))\]

\[\text{let } \text{FAC} = \text{Fix } \text{FAC-s}\]

The Y combinator discovered by Haskell Curry:

\[\text{let } \text{FIX } = \text{stp} \rightarrow (\lambda x \rightarrow \text{stp} (x \ x)) (\lambda x \rightarrow \text{stp} (x \ x))\]

↑ klop’s combinator
How does it work?

eval fix_step:

\[
\begin{align*}
\text{FIX STEP} & \triangleq d \rightarrow \left( \lambda x \rightarrow \text{stp}(x \ x) \right) \ \text{STEP} \\
& \triangleq b \rightarrow \left( \lambda x \rightarrow \text{STEP}(x \ x) \right) \ \text{STEP} \\
& = b \rightarrow \left( \lambda x \rightarrow \text{STEP}(x \ x) \right) \ \text{STEP} \\
& = b \rightarrow \text{STEP} \left( \lambda x \rightarrow \text{STEP}(x \ x) \right) \ \text{STEP} \\
& = \text{STEP} \left( \text{FIX STEP} \right)
\end{align*}
\]

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

**Next week:** We’ll look at the language named after him (Haskell)
progsys/liquidhaskell-blog/).