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

Alonzo Church

The Lambda Calculus
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

'define' "call"
No, really

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

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

- **Variable**
  - $x$, $y$, $z$

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

- **Application** (aka function call)
  - $e_1 \ e_2$
    - $e_1$ is the *function*, $e_2$ is the *argument*
    - in your favorite language: $e_1(e_2)$
(Here each of e, e1, e2 can itself be a variable, abstraction, or application)

**Examples**

\[
\lambda x \rightarrow x \quad -- \text{The identity function}
\]

\[
\lambda x \rightarrow (\lambda y \rightarrow y) \quad -- \text{A function that returns the identity function}
\]

\[
\lambda f \rightarrow f (\lambda x \rightarrow x) \quad -- \text{A function that applies its argument}
\]

\[
\quad \text{to the identity function}
\]
**QUIZ**

Which of the following terms are syntactically incorrect?

- **A.** \( (\lambda x \to x) \to y \)
- **B.** \( \lambda x \to \Box x \)
- **C.** \( \lambda x \to x \ (y \ x) \)
- **D.** A and C
- **E.** all of the above

*Correct answer: A*
Examples

\[ x \rightarrow x \]  -- The identity function
  -- ("for any x compute x")

\[ x \rightarrow (y \rightarrow y) \]  -- A function that returns the identity function

\[ f \rightarrow f (x \rightarrow x) \]  -- 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 \( y \)?
\( x \rightarrow (y \rightarrow y) \) -- A function that returns the identity function
-- OR: a function that takes two arguments
-- and returns the second one!

\[ ((e \text{ apple}) \text{ banana}) \rightarrow \text{ banana} \]

How do I apply a function to two arguments?

- e.g. apply \( x \rightarrow (y \rightarrow y) \) 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 -&gt; (\y -&gt; (\z -&gt; e))</td>
<td>\x -&gt; \y -&gt; \z -&gt; e</td>
</tr>
<tr>
<td>\x -&gt; \y -&gt; \z -&gt; e</td>
<td>\x y z -&gt; e</td>
</tr>
<tr>
<td>(((e1 e2) e3) e4)</td>
<td>e1 e2 e3 e4</td>
</tr>
</tbody>
</table>

\x y -> y -- *A function 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:

\[(x + 2) \times (3x - 1)\]

= ???

\[\downarrow\]

\[= 3 \times (4 - 5)\]

= ???

\[\downarrow\]

\[= 3 \times 1\]

= ???

\[\downarrow\]

\[= 3\]

---

**Execute** = rewrite step-by-step following simple rules, until no more rules apply
Rewrite Rules of Lambda Calculus

\[ \lambda x \to (\lambda x \to x) \]
\[ \lambda a \to (\lambda b \to b) \]

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

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 \( \lambda x \to e \)
• \( x \) is the newly introduced variable

• \( e \) is the scope of \( x \)

• any occurrence of \( x \) in \( \backslash x \rightarrow e \) is bound (by the binder \( \backslash x \))

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

\[
\begin{align*}
\backslash x \rightarrow x \\
\backslash x \rightarrow (\backslash y \rightarrow x)
\end{align*}
\]

An occurrence of \( x \) in \( e \) is free if it’s not bound by an enclosing abstraction

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

\[
\begin{align*}
&\quad x \ y \quad \quad \quad \text{-- no binders at all!} \\
&\quad \backslash y \rightarrow x \ y \quad \quad \quad \text{-- no \( \backslash x \) binder} \\
&\quad (\backslash x \rightarrow \backslash y \rightarrow y) \ x \quad \text{-- \( x \) is outside the scope of the \( \backslash x \) binder;} \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{-- intuition: it's not "the same" \( x \)}
\end{align*}
\]
In the expression $(x \rightarrow x) \times$, is $x$ bound or free?

A. bound
B. free
C. first occurrence is bound, second is free
D. first occurrence is bound, second and third are free
E. first two occurrences are bound, third is free

Correct answer: C
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:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(\lambda x \to e) &= FV(e) \setminus \{x\} \\
FV(e_1 e_2) &= FV(e_1) + FV(e_2)
\end{align*}
\]
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?

Answer: $\lambda x \to x$
Rewrite Rules of Lambda Calculus

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

Semantics: \(\beta\)-Reduction

\( (\lambda x \rightarrow e_1) e_2 \rightarrow b \rightarrow e_1[x := e_2] \)

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

Examples

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

=b> apple
Is this right? Ask Elsa (http://goto.ucsd.edu:8095/index.html#?demo=blank.lc)!

\((f \rightarrow f (x \rightarrow x)) (\text{give apple})\)
\=
\= b> \text{give apple} (x \rightarrow x)

**QUIZ**

\((x \rightarrow (y \rightarrow y)) \text{ apple}\)
\=
\= b> ???

A. apple
B. \( y \to \text{apple} \)
C. \( x \to \text{apple} \)
D. \( y \to y \)
E. \( x \to y \)

Correct answer: D.

QUIZ

(\( x \to x (\backslash x \to x) \)) \text{apple} 
=\text{b> ???}
A. apple (\x \rightarrow x)

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

C. apple (\text{x} \rightarrow \text{apple})

D. apple

E. \text{x} \rightarrow \text{x}

Correct answer: A.

\textit{A Tricky One}
Something is Fishy

\((x \to (y \to x)) \ y\)
\= \(b \to \ (y \to y)\)

Is this right?

**Problem:** the free \(y\) in the argument has been captured by \(\ y\)!
**Solution:** make sure that all *free variables* of the argument are different from the binders in the body.

---

**Capture-Avoiding Substitution**

We have to fix our definition of $\beta$-reduction:

$$ (\lambda x \to e_1) \ e_2 \ =_{b} \ e_1[x := e_2] $$

where $e_1[x := e_2]$ means "*e_1* with all *free* occurrences of $x$ replaced with $e_2$", as long as no free variables of $e_2$ get captured.

- *e_1* with all *free* occurrences of $x$ replaced with $e_2$, *as long as* no free variables of $e_2$ get captured
- undefined otherwise
Formally:

\[
\begin{align*}
x[x := e] &= e \\
y[x := e] &= y \quad \text{-- assuming } x \neq y \\
(e1 \ e2)[x := e] &= (e1[x := e]) \ (e2[x := e]) \\
(\x \to e1)[x := e] &= \x \to e1 \quad \text{-- why do we leave `e1` alone?} \\
(\y \to e1)[x := e] &= \\
\quad \mid \text{not (y in FV(e))} &= \y \to e1[x := e] \\
\quad \mid \text{otherwise} &= \text{undefined} \quad \text{-- wait, but what do we do the n???} \\
\end{align*}
\]

Answer: We leave e1 above alone even though it might contain x, because in \x \to e1 every occurrence of x is bound by \x (hence, there are no free occurrences of x)

**Rewrite Rules of Lambda Calculus**
1. $\alpha$-step (aka renaming formals)
2. $\beta$-step (aka function call)

**Semantics: $\alpha$-Renaming**

\[
\lambda x \rightarrow e \xrightarrow{\alpha} \lambda y \rightarrow e[x := y]
\]

where not (y in FV(e))

- We can rename a formal parameter and replace all its occurrences in the body
- We say that $\lambda x \rightarrow e$ $\alpha$-steps to $\lambda y \rightarrow e[x := y]$
Example:
\[ x \rightarrow x = a \rightarrow y \rightarrow y = a \rightarrow z \rightarrow z \]

All these expressions are \( \alpha \)-equivalent.

What’s wrong with these?

--- (A)
\[ f \rightarrow f \ x = a \rightarrow x \rightarrow x \ x \]

Answer: it violates the side-condition for \( \alpha \)-renaming that the new formal \( x \) must not occur freely in the body.

--- (B)
\[ (x \rightarrow y \rightarrow y) \ y = a \rightarrow (x \rightarrow z \rightarrow z) \ z \]

Answer: we should only rename within the body of the abstraction; the second \( y \) is a free variable, and hence must remain unchanged.

--- (C)
\[ x \rightarrow y \rightarrow x \ y = a \rightarrow \text{apple} \rightarrow \text{orange} \rightarrow \text{apple} \ \text{orange} \]

Answer: it’s fine, but technically it’s two \( \alpha \)-steps and not one.
The Tricky One

(\x \to (\y \to \x)) \ y
= a \to (\x \to (\z \to \x)) \ y
= b \to \ z \to \ y

To avoid getting confused, you can always rename formals, so that different variables have different names!
Normal Forms

A redex is a \( \lambda \)-term of the form

\( (\lambda x \rightarrow e_1) \ e_2 \)

A \( \lambda \)-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
C. (\(\langle x \rightarrow x\rangle\) y
D. x (\(\langle y \rightarrow y\rangle\)
E. C and D

Answer: C
Semantics: Evaluation

A \texttt{\lambda-}term \texttt{e} evaluates to \texttt{e'} if

1. There is a sequence of steps

\texttt{e =?> e_1 =?> ... =?> e_N =?> e'}

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

2. \texttt{e'} is in normal form

Examples of Evaluation

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

=\texttt{b}> \text{ apple}
\[ (\lambda f \cdot f (\lambda x \cdot x)) (\lambda x \cdot x) \]
\[ = b \cdot (\lambda x \cdot x) (\lambda x \cdot x) \]
\[ = b \cdot \lambda x \cdot x \]

\[ (\lambda x \cdot x x) (\lambda x \cdot x) \]
\[ = b \cdot (\lambda x \cdot x) (\lambda x \cdot x) \]
\[ = b \cdot \lambda x \cdot x \]

**Elsa shortcuts**

Named \( \lambda \)-terms:

\[ \text{let ID} = \lambda x \cdot x \quad -- \text{abbreviation for} \ \lambda x \cdot x \]

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

What is the difference?
Non-Terminating Evaluation

\((\lambda x \rightarrow x \, x) \, (\lambda x \rightarrow x \, x)\)
\[= b > (\lambda x \rightarrow x \, x) \, (\lambda x \rightarrow x \, x)\]

Oops, we can write programs that loop back to themselves...

and never reduce to a normal form!

This combinator is called \(\Omega\)

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

```
let OMEGA = (\x \rightarrow x \, x) \, (\x \rightarrow x \, x)
(\x \rightarrow \lambda y \rightarrow y) \, 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
- Functions [we got those]
- Recursion

Let's see how to encode all of these features with the $\lambda$-calculus.
**λ-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*

- if b then e1 else e2
Booleans: API

We need to define three functions

\[
\text{let } \text{TRUE} = ??? \\
\text{let } \text{FALSE} = ??? \\
\text{let } \text{ITE} = \lambda b \times y \rightarrow ??? \quad \text{-- if } b \text{ then } x \text{ else } y
\]

such that

\[
\text{ITE TRUE apple banana} = \rightarrow \text{apple} \\
\text{ITE FALSE apple banana} = \rightarrow \text{banana}
\]

(Here, let NAME = e means NAME is an abbreviation for e)
**Booleans: Implementation**

```plaintext
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**
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
=d> (\b x y -> b x y) FALSE e1 e2  -- expand def ITE
=b> (\x y -> FALSE x y) e1 e2  -- beta-step
=b> (\y -> FALSE e1 y) e2  -- beta-step
=b> FALSE e1 e2  -- expand def FALSE
=d> (\x y -> y) e1 e2  -- beta-step
=b> (\y -> y) e2  -- beta-step
=b> e2

Boolean Operators

Now that we have ITE it’s easy to define other Boolean operators:
let NOT = \b       - > ???

let AND = \b1 b2 - > ???

let OR   = \b1 b2 - > ???
\textbf{let} NOT = \( \backslash b \rightarrow \text{ITE } b \text{ FALSE TRUE} \)

\textbf{let} AND = \( \backslash b_1 \ b_2 \rightarrow \text{ITE } b_1 \ b_2 \text{ FALSE} \)

\textbf{let} OR = \( \backslash b_1 \ b_2 \rightarrow \text{ITE } b_1 \text{ TRUE } b_2 \)

Or, since ITE is redundant:

\textbf{let} NOT = \( \backslash b \rightarrow b \text{ FALSE TRUE} \)

\textbf{let} AND = \( \backslash b_1 \ b_2 \rightarrow b_1 \ b_2 \text{ FALSE} \)

\textbf{let} OR = \( \backslash b_1 \ b_2 \rightarrow b_1 \text{ TRUE } b_2 \)

\textit{Which definition do you prefer and why?}
Programming in λ-calculus

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

FST (PAIR apple banana) => apple
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 $y$)

```plaintext
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?

```plaintext
let TRIPLE = \x y z -> PAIR x (PAIR y z)
let FST3 = \t -> FST t
let SND3 = \t -> FST (SND t)
let TRD3 = \t -> SND (SND t)
```
Programming in λ-calculus

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

\[
\begin{align*}
\text{IS}_{\text{ZERO}} \text{ ZERO} & \Rightarrow \text{TRUE} \\
\text{IS}_{\text{ZERO}} (\text{INC} \text{ ZERO}) & \Rightarrow \text{FALSE} \\
\text{INC} \text{ ONE} & \Rightarrow \text{TWO} \\
\ldots
\end{align*}
\]
Natural Numbers: Implementation

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

\[
\begin{align*}
\text{let } \text{ONE} & = \lambda x . f x \\
\text{let } \text{TWO} & = \lambda x . f (f x) \\
\text{let } \text{THREE} & = \lambda x . f (f (f x)) \\
\text{let } \text{FOUR} & = \lambda x . f (f (f (f x))) \\
\text{let } \text{FIVE} & = \lambda x . f (f (f (f (f x)))) \\
\text{let } \text{SIX} & = \lambda x . f (f (f (f (f (f x))))) \\
\end{align*}
\]

\[ \ldots \]

QUIZ: Church Numerals
Which of these is a valid encoding of ZERO?

- **A**: let ZERO = \(f \ x \rightarrow x\)
- **B**: let ZERO = \(f \ x \rightarrow f\)
- **C**: let ZERO = \(f \ x \rightarrow f \ x\)
- **D**: let ZERO = \(x \rightarrow x\)
- **E**: None of the above

*Answer: A*

Does this function look familiar?

*Answer: It’s the same as FALSE!*
$\lambda$-calculus: Increment

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

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)

Answer: A

\[\text{\textbf{\textit{\lambda-calculus: Addition}}}

\text{-- Call \textit{f} on \textit{x} exactly \textit{n + m} times}

\textbf{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
Answer: B or C

\[ \text{\textit{\lambda-calculus: Multiplication}} \]

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

\[
\texttt{let } \text{MUL} = \lambda n \ m \rightarrow n \ (\text{ADD} \ m) \ \text{ZERO}
\]

Example:

\[
\text{eval two\_times\_three :}
\]
\[
\text{MUL \ TWO \ ONE}
\]
\[
\Rightarrow \ TWO
\]
Programming in $\lambda$-calculus

- Booleans [done]
- Records (structs, tuples) [done]
- Numbers [done]
- Functions [we got those]
- Recursion
\[ \lambda\text{-calculus: Recursion} \]

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

\[ n \rightarrow \ldots \quad \text{-- } 1 + 2 + \ldots + n \]

**QUIZ**

Is this a correct implementation of \textit{SUM}?
\textbf{let} \textit{SUM} = \lambda n \to \textit{ITE} (\textit{ISZ} \; n) \\
\quad \text{ZERO} \\
\quad \text{(ADD} \; n \; \text{(SUM} \; (\text{DEC} \; n))) \\

\begin{enumerate}
\item Yes
\item No
\end{enumerate}

\begin{itemize}
\item Named terms in Elsa are just syntactic sugar
\item To translate an Elsa term to \(\lambda\)-calculus: replace each name with its definition
\end{itemize}

\\textit{\begin{verbatim}
\lambda n \to \textit{ITE} (\textit{ISZ} \; n) \\
\quad \text{ZERO} \\
\quad \text{(ADD} \; n \; \text{(SUM} \; (\text{DEC} \; n)))  \quad \textit{-- But SUM is not a thing!}
\end{verbatim}\end{itemize}
Recursion:

- Inside this function I want to call the same function on $DEC\ n$

Looks like we can’t do recursion, because it requires being able to refer to functions by name, but in $\lambda$-calculus functions are anonymous.

Right?

$\lambda$-calculus: Recursion

Think again!
Recursion:

- Inside this function I want to call the same function on DEC n
- Inside this function I want to call a function on DEC n
- And BTW, I want it 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 something clever to STEP, so that the function passed as rec itself becomes

```
\n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))
```
\textbf{\lambda-calculus: Fixpoint Combinator}

\textbf{Wanted:} a combinator \texttt{FIX} such that \texttt{FIX \ STEP} calls \texttt{STEP} with itself as the first argument:

\begin{verbatim}
 FIX STEP 
 =>*> STEP (FIX STEP)
\end{verbatim}

(In math: a \textit{fixpoint} of a function $f(x)$ is a point $x$, such that $f(x) = x$)

Once we have it, we can define:

\begin{verbatim}
 let SUM = FIX STEP
\end{verbatim}
Then by property of \( \text{FIX} \) we have:

\[
\text{SUM} \Rightarrow^* \text{STEP SUM} \quad \text{- (1)}
\]

\text{eval sum\_one:}

\[
\text{SUM ONE} =
\Rightarrow^* \text{STEP SUM ONE} \quad \text{- (1)}
\]

\[
d\Rightarrow \left( \\text{rec n} \rightarrow \text{ITE (ISZ n) ZERO (ADD n (rec (DEC n)))} \right) \text{SUM ONE}
\]

\[
b\Rightarrow \left( \\text{n} \rightarrow \text{ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))} \right) \text{ONE}
\]

\[
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{-- ^^ the magic happened!}
\]

\[
b\Rightarrow \text{ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE)))}
\]

\[
\Rightarrow^* \text{ADD ONE (SUM ZERO)} \quad \text{- \textit{def of ISZ, ITE, DEC, ...}}
\]

\[
\Rightarrow^* \text{ADD ONE (STEP SUM ZERO)} \quad \text{- (1)}
\]

\[
d\Rightarrow \text{ADD ONE}
\]

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

\[
b\Rightarrow \text{ADD ONE} \left( \left( \text{n} \rightarrow \text{ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))} \right) \text{ZERO} \right)
\]

\[
b\Rightarrow \text{ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM (DEC ZERO))))}
\]

\[
b\Rightarrow \text{ADD ONE ZERO}
\]

\[
\Rightarrow \text{ONE}
\]

How should we define \( \text{FIX} \)???
The Y combinator

Remember $\Omega$?

$$(\lambda x \to x \, x) \, (\lambda x \to x \, x)$$

= $$(\lambda x \to x \, x) \, (\lambda x \to x \, x)$$

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

The Y combinator discovered by Haskell Curry:

```haskell
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 ^^^^^^^^^^