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
  - ... and more

Which ones can we do without?

What is the smallest universal language?
What is computable?

- Prior to 1930s
  - Informal notion of an effectively calculable function:
    One that can be computed by a human with pen and paper, following an algorithm

What is computable?

- 1936: Formalization

  Alan Turing: Turing machines

  $0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1 \ 8 \ 0 \ 0$

  $q_0$

What is computable?

- 1936: Formalization

  Alonzo Church: lambda calculus

  $e ::= x$
  $| \ x -> e$
  $| e1 \ e2$
The Next 700 Languages

- Big impact on language design!

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

Peter Landin, 1966

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
  - … and more

The Lambda Calculus

- Features
  - Functions
  - (that’s it)
The Lambda Calculus

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

The 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

\[ e ::= x \mid \lambda x \to e \mid e_1 e_2 \]

- Programs are expressions \( e \) (also called \( \lambda \)-terms)
- Variable: \( x, y, z \)
- Abstraction (aka nameless function definition):
  - \( \lambda x \to e \) “for any \( x \), compute \( e \)”
  - \( x \) is the formal parameter, \( e \) is the body
- Application (aka function call):
  - \( e_1 e_2 \) “apply \( e_1 \) to \( e_2 \)”
  - \( e_1 \) is the function, \( e_2 \) is the argument
Examples

-- The identity function ("for any x compute x")
\( \lambda x \to x \)

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

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

QUIZ: Lambda syntax

Which of the following terms are syntactically incorrect? *

- A. \( \lambda (x \to x) \to y \)
- B. \( \lambda x \to x x \)
- C. \( \lambda x \to (y x) \)
- A and C
- All of the above

http://tiny.cc/cmps112-lambda-ind
Examples

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

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

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

- How do I define a function with two arguments?
  - e.g. a function that takes x and y and returns y

Examples

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

OR: a function that takes two arguments
and returns the second one!

- How do I define a function with two arguments?
  - e.g. a function that takes x and y and returns y

Examples

- How do I apply a function to two arguments?
  - e.g. apply \x \rightarrow (\y \rightarrow y) to apple and banana?

-- first apply to apple, then apply the result to banana

(((\x \rightarrow (\y \rightarrow y)) apple) banana)
**Syntactic Sugar**

- Convenient notation used as a shorthand for valid syntax

<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 \rightarrow y \rightarrow z \rightarrow e )</td>
</tr>
<tr>
<td>( ((e_1 \ e_2) \ e_3) \ e_4 )</td>
<td>( e_1 \ e_2 \ e_3 \ e_4 )</td>
</tr>
</tbody>
</table>

\( x \rightarrow y \) -- *A function that takes two arguments*  
-- and returns the second one...

\( (x \rightarrow y) \) apple banana -- ... *applied to two arguments*

---

**Semantics: What programs mean**

- How do I “run” or “execute” a \( \lambda \)-term?

- Think of middle-school algebra:
  
  -- *Simplify expression:*
  
  \( (x + 2) \ast (3x - 1) \)
  
  =
  
  ???

  **Execute** = rewrite step-by-step following simple rules until no more rules apply

---

**Rewrite rules of lambda calculus**

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 \( \lambda x \to e \) is bound (by the binder \( \lambda x \))

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

\[
\lambda x \to x \\
\lambda x \to (\lambda y \to x)
\]

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

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

\[
x \ y 
\]
\[
\to \ x \ y 
\]
\[
\to (\lambda y \to x) \ x 
\]

**intuition:** it’s not “the same” \( x \)

QUIZ: Variable scope

In the expression \( (\lambda x \to x) x \), 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

http://tiny.cc/cmbs112-scope-ind
QUIZ: Variable scope

In the expression \((x \rightarrow x) x\), 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

http://tiny.cc/cmps112-scope-grp

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:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(x \rightarrow e) &= FV(e) \setminus \{x\} \\
FV(e_1, e_2) &= FV(e_1) \cup FV(e_2)
\end{align*}
\]
Closed Expressions

- If e has no free variables it is said to be closed
- Closed expressions are also called combinators
  - Q: What is the shortest closed expression?
  - A: \x -> x

Rewrite rules of lambda calculus

1. α-step (aka renaming formals)
2. β-step (aka function call)
Semantics: Β-Reduction

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

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

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

Is this right? Ask Elsa!

\((\lambda f \to f (\lambda x \to x)) (\text{give apple}) \rightarrow b \to ???\)

Examples

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

Is this right? Ask Elsa!

\((\lambda f \to f (\lambda x \to x)) (\text{give apple}) \rightarrow b \to \text{give apple} (\lambda x \to x)\)
QUIZ: β-Reduction 1

\((λx \to (y \to y))\) apple =b> ??? *

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

http://tiny.cc/cmps112-beta1-ind

QUIZ: β-Reduction 1

\((λx \to (y \to y))\) apple =b> ??? *

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

http://tiny.cc/cmps112-beta1-grp

QUIZ: β-Reduction 2

\((λx \to x (λx \to x))\) apple =b> ??? *

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

http://tiny.cc/cmps112-beta2-ind
QUIZ: β-Reduction 2

\((x \to x (x \to x)) \text{ apple} =_β ?? \star\)

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

http://tiny.cc/cmps112-beta2-grp

---

A Tricky One

\((x \to (y \to x))\ y =_β \ y \to y\)

Is this right?

**Problem:** the free y in the argument has been captured by \(\lambda y\)!

**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 β-reduction:

\((\lambda x \to e_1) \ e_2 =_β e_1[x := e_2]\)

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
- undefined otherwise
Capture-Avoiding Substitution

Formally:

\[ \begin{align*} 
   x[x := e] &= e \\
   y[x := e] &= y \quad \text{-- assuming } x \neq y \\
   (e_1 e_2)[x := e] &= (e_1[x := e])(e_2[x := e]) \\
   (\lambda x \to e_1)[x := e] &= \lambda x \to e_1 \quad \text{-- why just } e_1? \\
   (\lambda y \to e_1)[x := e] &\mid \text{not } (y \in \text{FV}(e)) = \lambda y \to e_1[x := e] \\
   &\mid \text{otherwise} = \text{undefined} \quad \text{-- but what then??} 
\end{align*} \]

Rewrite rules of lambda calculus

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

Semantics: \(\alpha\)-Reduction

\(\lambda x \to e \quad =_{\alpha} \quad \lambda y \to e[x := y]
\text{where not } (y \in \text{FV}(e))

- We can rename a formal parameter and replace all its occurrences in the body
- We say that \((\lambda x \to e) \quad \alpha\text{-steps} \quad \text{to } e[x := y]\)
Semantics: $\alpha$-Reduction

\[
\begin{align*}
  \lambda x \rightarrow e \rightarrow a & \rightarrow \lambda y \rightarrow e[x := y] \\
  \text{where not } (y \in \text{FV}(e))
\end{align*}
\]

- Example:
  \[
  \begin{align*}
  \lambda x \rightarrow x & \rightarrow a \rightarrow \lambda y \rightarrow y \rightarrow a \rightarrow \lambda z \rightarrow z
  \end{align*}
  \]
  - All these expressions are $\alpha$-equivalent

Example

What's wrong with these?

-- (A)
\[
\begin{align*}
  \lambda f \rightarrow f x & \rightarrow a \rightarrow \lambda x \rightarrow x
  \end{align*}
\]

-- (B)
\[
\begin{align*}
  (\lambda x \rightarrow \lambda y \rightarrow y) y & \rightarrow a \rightarrow (\lambda x \rightarrow \lambda z \rightarrow z) z
  \end{align*}
\]

-- (C)
\[
\begin{align*}
  \lambda x \rightarrow \lambda y \rightarrow x y & \rightarrow a \rightarrow \lambda \text{apple} \rightarrow \lambda \text{orange} \rightarrow \text{apple orange}
  \end{align*}
\]

The Tricky One

\[
\begin{align*}
  (\lambda x \rightarrow (\lambda y \rightarrow x)) y \\
  \rightarrow a \rightarrow ???
  \end{align*}
\]

To avoid getting confused, you can always rename formals, so that different variables have different names!
The Tricky One

\( (\lambda x \rightarrow (\lambda y \rightarrow x)) \ y \)
= \( a \rightarrow (\lambda x \rightarrow (\lambda z \rightarrow x)) \ y \)
= \( b \rightarrow \lambda z \rightarrow y \)

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: Normal form

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

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

http://tiny.cc/cmps112-norm-ind
QUIZ: Normal form

Which of the following terms are not in normal form?

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

http://tiny.cc/cmps112-norm-grp

Semantics: Evaluation

- A \lambda\text{-}term e evaluates to e’ if
  1. There is a sequence of stops
     
     e =?> e_1 =?> ... =?> e_N =?> e’

     where each =?> is either =a> or =b> and N >= 0
  2. e’ is in normal form

Example of evaluation

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

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

(\x -> x) (\x -> x)
=?> ???
Example of evaluation

\( (\lambda x \to x) \text{apple} \)
\[ \text{b} \to \text{apple} \]

\( (\lambda f \to f (\lambda x \to x)) \ (\lambda x \to x) \)
\[ \text{b} \to (\lambda x \to x) \ (\lambda x \to x) \]
\[ \text{b} \to (\lambda x \to x) \]

\( (\lambda x \to x \ x) \ (\lambda x \to x) \)
\[ \Rightarrow ??? \]

Example of evaluation

\( (\lambda x \to x) \text{apple} \)
\[ \text{b} \to \text{apple} \]

\( (\lambda f \to f (\lambda x \to x)) \ (\lambda x \to x) \)
\[ \text{b} \to (\lambda x \to x) \ (\lambda x \to x) \]
\[ \text{b} \to (\lambda x \to x) \]

\( (\lambda x \to x \ x) \ (\lambda x \to x) \)
\[ \Rightarrow ??? \]

Elsa shortcuts

- Named \( \lambda \)-terms
  
  \[ \text{let ID} = \lambda x \to x -- \text{abbreviation for} \ (\lambda x \to x) \]

- To substitute a name with its definition, use a \( \Rightarrow \) step:
  
  \[ \text{ID apple} \]
  \[ \Rightarrow (\lambda x \to x) \text{apple} -- \text{expand definition} \]
  \[ \Rightarrow \text{apple} -- \text{beta-reduce} \]
Elsa shortcuts

• 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 \( e_2 \)

• What is the difference?

Non-Terminating Evaluation

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

• Oh no... we can write programs that loop back to themselves
• And never reduce to normal form!
• This combinator is called \( \Omega \)

Non-Terminating Evaluation

• What if we pass \( \Omega \) 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 $\lambda$-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.

$\lambda$-calculus: Booleans

- How can we encode Boolean values (TRUE and FALSE) as functions?
- Well, what do we do with a Boolean $b$?
  - We make a binary choice
    \[
    \text{if } b \text{ then } e_1 \text{ else } e_2
    \]

Booleans: API

- We need to define three functions
  \[
  \begin{align*}
  \text{let } \text{TRUE} &= ??? \\
  \text{let } \text{FALSE} &= ??? \\
  \text{let } \text{ITE} &= \lambda b \times y \rightarrow ??? \quad \text{-- if } b \text{ then } x \text{ else } y
  \end{align*}
  \]

such that

\[
\begin{align*}
\text{ITE \ TRUE \ apple \ banana} &= \rightarrow \text{apple} \\
\text{ITE \ FALSE \ apple \ banana} &= \rightarrow \text{banana}
\end{align*}
\]

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

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

Example: Branches step-by-step

```plaintext
eval ITE_true:
ITE TRUE e1 e2
  = \x y -> b x y TRUE e1 e2 -- expand def ITE
  = \x y -> TRUE x y e1 e2 -- beta-step
  = \y -> TRUE e1 y e2 -- beta-step
  = TRUE e1 e2 -- expand def TRUE
  = \x y -> x e1 e2 -- beta-step
  = e1 e2 -- beta-step
  = 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](http://goto.ucsd.edu:8095/index.html#?demo=ite.lc)

```plaintext
eval ITE_false:
ITE FALSE e1 e2
  -- fill the steps in!
  = e2
```
Example: Branches step-by-step

```plaintext
eval ite_false:
ITE FALSE e1 e2
  \b (\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 TRUE
  \b (\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:

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

Boolean operators

- Now that we have ITE it’s easy to define other Boolean operators:

```plaintext
let NOT = \b -> ITE b FALSE TRUE
let AND = \b1 b2 -> ITE b1 b2 FALSE
let OR = \b1 b2 -> ITE b1 TRUE b2
```
Boolean operators

- Now that we have ITE it’s easy to define other Boolean operators:

```plaintext
let NOT = \b -> b FALSE TRUE
let AND = \b1 b2 -> b1 b2 FALSE
let OR = \b1 b2 -> b1 TRUE b2
```

- (since ITE is redundant)
- *Which definition do you prefer and why?*

Programming in λ-calculus

- Real languages have lots of features
  - *Booleans* [done]
  - Records (structs, tuples)
  - Numbers
  - *Functions* [we got those]
  - Recursion

λ-calculus: Records

- Let’s start with records with two fields (aka pairs)?
- Well, 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

\[
\text{let } \text{PAIR} = \lambda x \ y \rightarrow \ ???? \quad \text{-- Make a pair with } x \text{ and } y \\
\text{let } \text{FST} = \lambda p \rightarrow ??? \quad \text{-- Return first element} \\
\text{let } \text{SND} = \lambda p \rightarrow ??? \quad \text{-- Return second element}
\]

\[
\{ \text{fst} : x, \text{snd} : y \}
\]

\[
\text{such that}
\]

\[
\text{FST(PAIR apple banana)} \rightarrow \text{apple} \\
\text{SND(PAIR apple banana)} \rightarrow \text{banana}
\]

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 \(y\))

\[
\text{let } \text{PAIR} = \lambda x \ y \rightarrow \{ \b \rightarrow \text{ITE } b \ x \ y \}
\]

\[
\text{let } \text{FST} = \lambda p \rightarrow p \text{ TRUE} \quad \text{-- call w/ TRUE, get 1st value}
\]

\[
\text{let } \text{SND} = \lambda p \rightarrow p \text{ FALSE} \quad \text{-- call w/ FALSE, get 2nd value}
\]

Exercise: Triples?

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

\[
\text{let } \text{TRIPLE} = \lambda x \ y \ z \rightarrow ???
\]

\[
\text{let } \text{FST3} = \lambda t \rightarrow ???
\]

\[
\text{let } \text{SND3} = \lambda t \rightarrow ???
\]

\[
\text{let } \text{TRD3} = \lambda t \rightarrow ???
\]
Exercise: Triples?

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

```latex
let TRIPLE = \(x \ y \ z \to \) PAIR x (PAIR y z)
let FST3 = \(t \to \) FST t
let SND3 = \(t \to \) FST (SND t)
let TRD3 = \(t \to \) SND (SND t)
```

Programming in $\lambda$-calculus

- Real languages have lots of features
  - Booleans [done]
  - Records (structs, tuples) [done]
  - Numbers
  - Functions [we got those]
  - Recursion

$\lambda$-calculus: Numbers

- Let’s start with natural numbers (0, 1, 2, ...)
- What do we do with natural numbers?

1. Count: 0, inc
2. Arithmetic: dec, +, -, *
3. 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.

\[
\begin{align*}
\text{IS_ZERO ZERO} & \implies \text{TRUE} \\
\text{IS_ZERO (INC ZERO)} & \implies \text{FALSE} \\
\text{INC ONE} & \implies \text{TWO} \\
& \ldots
\end{align*}
\]

Pairs: Implementation

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

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

\text{\( \lambda \)-calculus: Increment}

\text{-- Call \( f \) on \( x \): one more time than \( n \) does}
\[
\text{let INC} \quad = \lambda n \mapsto (\lambda x \mapsto ???)
\]

- Example

\[
\begin{align*}
\text{eval inc zero :} \\
\text{INC ZERO} & \implies (\lambda f \mapsto f \ (n \ f \ x)) \ \text{ZERO} \\
& \implies (\lambda x \mapsto f \ (\text{ZERO} \ f \ x)) \\
& \implies (\lambda x \mapsto f \ x) \\
& \implies \text{ONE}
\end{align*}
\]
**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
```

**λ-calculus: Multiplication**

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

• Example

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

**Programming in λ-calculus**

• Real languages have lots of features
  - Booleans [done]
  - Records (structures, tuples) [done]
  - Numbers [done]
  - Functions [we got those]
  - Recursion
\textbf{\(\lambda\)-calculus: Recursion}

- I want to write a function that sums up natural numbers up to \(n\):
  \[
  \lambda \rightarrow \ldots \quad \text{-- } 1 + 2 + \ldots + n
  \]

- \textbf{No!} Named terms in Elsa are just syntactic sugar
- To translate an Elsa term to \(\lambda\)-calculus: replace each name with its definition
  \[
  \lambda \rightarrow \text{ITE (ISZ }n) \text{ ZERO (ADD }n (\text{SUM (DEC }n))) \quad \text{-- But SUM is not a thing!}
  \]

- \textbf{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 \textit{by name}, but in \(\lambda\)-calculus functions are \textit{anonymous}.
  - \textbf{Right?}

\textbf{\(\lambda\)-calculus: Recursion}

- Think again!
- \textbf{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
- \textbf{Step 1:} Pass in the function to call “recursively”

  \[
  \text{let } \text{STEP }= \\
  \lambda \rightarrow \text{ITE (ISZ }n) \text{ ZERO (ADD }n (\text{rec (DEC }n))) \quad \text{-- Call some rec}
  \]

- Right?
**λ-calculus: Recursion**

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

```plaintext
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

```plaintext
\n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))
```

**λ-calculus: Fixpoint Combinator**

- **Wanted**: a combinator `FIX` such that `FIX STEP` calls `STEP` with itself as the first argument:

```plaintext
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:

```plaintext
let SUM = FIX STEP
```

- Then by property of `FIX` we have:

```plaintext
SUM =*> STEP SUM -- (1)
```

**λ-calculus: Fixpoint Combinator**

**eval sum_one:**

```plaintext
eval sum_one:
  SUM ONE
  =*> STEP SUM ONE -- (1)
  =d ((\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ONE
  =b> (\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ONE
  -- ^^ the magic happened!
  =b> ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE)))
  =*> ADD ONE (SUM ZERO) -- def of ISZ, ITE, DEC, ...
  =*> ADD ONE (STEP SUM ZERO) -- (1)
  =d> ADD ONE
  (((\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ZERO)
  =b> ADD ONE (((\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ZERO)
  =b> ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM (DEC ZERO))))
  =b> ADD ONE ZERO
  ==> ONE
```
\(\lambda\)-calculus: Fixpoint Combinator

- So how do we define \(\text{FIX}\)?

- Remember \(\Omega\)? It \textit{replicates itself}!
  \[
  (\lambda x \to x \to x) \ (\lambda x \to x \to x) \\
  = b > (\lambda x \to x \to x) \ (\lambda x \to x \to x)
  \]

- We need something similar but more involved.

\(\lambda\)-calculus: Fixpoint Combinator

- The \(Y\) combinator discovered by Haskell Curry:
  \[
  \text{let } \text{FIX} = \lambda \text{stp} \to (\lambda x \to \text{stp} \ (x \to x)) \ (\lambda x \to \text{stp} \ (x \to x))
  \]

- How does it work?
  
  \[
  \text{eval } \text{fix\_step}:
  \]

  fix\_step
  = \text{FIX } \text{STEP} \\
  = d > (\lambda x \to \text{stp} \ (x \to x)) \ (\lambda x \to \text{stp} \ (x \to x)) \ \text{STEP} \\
  = b > (\lambda x \to \text{STEP} \ (x \to x)) \ (\lambda x \to \text{STEP} \ (x \to x)) \\
  = b > \text{STEP} ((\lambda x \to \text{STEP} \ (x \to x)) \ (\lambda x \to \text{STEP} \ (x \to x))) \\
  = -- \ ^^^^^^^^^ this \ is \ \text{FIX } \text{STEP} \ ^^^^^^^^^
  
Programming in \(\lambda\)-calculus

- Real languages have lots of features
  - Booleans [done]
  - Records (structs, tuples) [done]
  - Numbers [done]
  - Functions [we got those]
  - Recursion [done]
Next time: Intro to Haskell