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

  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
  \x -> e
  e1 e2
```

- Programs are expressions e (also called λ-terms)
- Variable: x, y, z
- Abstraction (aka nameless function definition):
  - \x -> e “for any x, compute e”
  - x is the formal parameter, e is the body
- Application (aka function call):
  - e1 e2 “apply e1 to e2”
  - e1 is the function, e2 is the argument
Examples

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

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

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

QUIZ: Lambda syntax

Which of the following terms are syntactically incorrect? *

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

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

QUIZ: Lambda syntax

Which of the following terms are syntactically incorrect? *

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

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

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

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

-- A function that applies its argument to
-- the identity function
\f -> f (\x -> 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 -> (\y -> 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 -> (\y -> y) to apple and banana?

-- first apply to apple, then apply the result to banana
(((\x -> (\y -> 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 \to (y \to (z \to e)) )</td>
<td>( x \to y \to z \to e )</td>
</tr>
<tr>
<td>( x \to y \to z \to e )</td>
<td>( x \to y \to z \to e )</td>
</tr>
<tr>
<td>( ((e1 \ e2) \ e3) \ e4 )</td>
<td>( e1 \ e2 \ e3 \ e4 )</td>
</tr>
</tbody>
</table>

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

\( (\lambda y \to y) \) 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:
  \[(x + 2)(3x - 1)\]
  \[= \]
  \[??\]

- 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 $\lambda x$)

Semantics: Scope of a Variable

- 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:
  \[
  x \ y \quad -- \text{no binders at all!} \\
  \lambda y \to x \ y \quad -- \text{no $\lambda x$ binder} \\
  (\lambda x \to \lambda y \to y) \ x \quad -- \text{x is outside the scope} \\
  \quad -- \text{of the $\lambda x$ binder;} \\
  \quad -- \text{intuition: it's not "the same" x}
  \]

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/cmps112-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:

\[
\begin{align*}
\text{FV}(x) &= \{x\} \\
\text{FV}(\lambda x \rightarrow e) &= \text{FV}(e) \setminus \{x\} \\
\text{FV}(e_1 \cdot e_2) &= \text{FV}(e_1) \cup \text{FV}(e_2)
\end{align*}
\]
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

\((\lambda x \to e_1) \ e_2 \ \mathsf{=} \to \ e_1[x := e_2]\)

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 \((\lambda x \to e_1) \ e_2\) β-steps to \(e_1[x := e_2]\)

Examples

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

Is this right? Ask Elsa!

\((\lambda f \to f \ (\lambda x \to x)) \ (\text{give apple})\)
=⇒ ???

Examples

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

Is this right? Ask Elsa!

\((\lambda f \to f \ (\lambda x \to x)) \ (\text{give apple})\)
=⇒ \text{give apple} (\lambda x \to x)
QUIZ: B-Reduction 1

\( (\lambda x \cdot (\lambda y \cdot y)) \) \( \text{apple} \Rightarrow \) ???

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

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

QUIZ: B-Reduction 1

\( (\lambda x \cdot (\lambda y \cdot y)) \) \( \text{apple} \Rightarrow \) ???

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

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

QUIZ: B-Reduction 2

\( (\lambda x \cdot (\lambda x \cdot \text{x})) \) \( \text{apple} \Rightarrow \) ???

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

http://tiny.cc/cmps112-beta2-ind
QUIZ: \( \beta \)-Reduction 2

\[
(\lambda x \cdot (\lambda x \cdot x)) \text{ apple} \rightarrow \ ??? \ \\
\]

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

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

A Tricky One

\[
(\lambda x \cdot (\lambda y \cdot x)) \ y \rightarrow \ ??? \ \\
\]

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 \( \beta \)-reduction:

\[
(\lambda x \cdot e1) \ e2 \rightarrow e1[x := e2] \\
\]

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
- undefined otherwise
Capture-Avoiding Substitution

Formally:

\[ x[x := e] = e \]
\[ y[x := e] = y -- assuming x /\= y \]
\[ (e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e]) \]
\[ (\lambda x \to e_1)[x := e] = \lambda x \to e_1 -- why just `e_1`? \]
\[ (\lambda y \to e_1)[x := e] \]
\[ | not (y \in \text{FV}(e)) = \lambda y \to e_1[x := e] \]
\[ | otherwise = \text{undefined} -- \text{but what then??} \]

Rewrite rules of lambda calculus

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

Semantics: \(\alpha\)-Reduction

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

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

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

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

### Example

What’s wrong with these?

- **(A)**
  \[
  \text{f -> f x} = \text{a} \quad \text{x -> x x}
  \]

- **(B)**
  \[
  (\text{x -> y -> y}) \text{ y} = \text{a} \quad (\text{x -> z -> z}) \text{ z}
  \]

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

### The Tricky One

\[
(\text{x -> (y -> x)}) \text{ y} = \text{a} \quad ???
\]

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

\((\lambda x \to (\lambda y \to x)) \, y\)

= \(a\) \((\lambda x \to (\lambda z \to x)) \, y\)

= \(b\) \((\lambda z \to y)\)

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

Normal Forms

A redex is a λ-term of the form

\((\lambda x \to e_1) \, e_2\)

A λ-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. \((\lambda x \to x) \, y\)
- D. \(x \,(y \to y)\)
- 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. xy
- C. (\x -> x) y
- D. x (y -> y)
- E. C and D

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

---

**Semantics: Evaluation**

- A λ-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

(\x -> f (\x -> x)) (\x -> x)

=?> ???

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

=?> ???
Example of evaluation

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

\( x \rightarrow x \) apple

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

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

\( x \rightarrow x \)

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

\( x \rightarrow x \)

Example of evaluation

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

\( x \rightarrow x \) apple

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

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

\( x \rightarrow x \)

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

\( x \rightarrow x \)

Elsa shortcuts

- Named \( \lambda \)-terms
  
  \texttt{let ID = x \rightarrow x} \quad \textit{abbreviation for} \quad \texttt{x \rightarrow x}

- To substitute a name with its definition, use a \( \rightarrow d \) step:
  
  \texttt{ID apple}
  
  \( =d\rightarrow (\lambda x \rightarrow x) \) apple \quad \textit{expand definition}
  
  \( =b\rightarrow \) apple \quad \textit{beta-reduce}
Elsa shortcuts

- Evaluation
  - $e_1 \Rightarrow^* e_2$: $e_1$ reduces to $e_2$ in 0 or more steps
    - where each step is $\Rightarrow a$, $\Rightarrow b$, or $\Rightarrow d$
  - $e_1 \Rightarrow e_2$: $e_1$ evaluates to $e_2$
- What is the difference?

Non-Terminating Evaluation

\[
(\lambda x \rightarrow x) (\lambda x \rightarrow x) \\
= \Rightarrow b (\lambda x \rightarrow x) (\lambda x \rightarrow x)
\]

- 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?
  
  \[
  \text{let } OMEGA = (\lambda x \rightarrow x) (\lambda x \rightarrow x) \\
  (\lambda x \rightarrow y \rightarrow y) OMEGA
  \]
- 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.

$\lambda$-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
  \[
  \begin{align*}
  \text{let } \text{TRUE} & = ??? \\
  \text{let } \text{FALSE} & = ??? \\
  \text{let } \text{ITE} & = \lambda b x y \to ??? \quad \text{-- if } b \text{ then } x \text{ else } y
  \end{align*}
  \]
  such that
  \[
  \begin{align*}
  \text{ITE } \text{TRUE} \text{ apple banana} & \to \text{apple} \\
  \text{ITE } \text{FALSE} \text{ apple banana} & \to \text{banana}
  \end{align*}
  \]
  (Here, let NAME = e means NAME is an abbreviation for e)
Booleans: Implementation

```latex
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

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

```latex
eval ITE_false:
ITE FALSE e1 e2
= \x y. (b x y -> b x y) FALSE e1 e2  -- expand def ITE
= \x y. (\y. TRUE e1 y) e1 e2  -- beta-step
= \x y -> e1) e2  -- beta-step
= e2
```

Now you try it!
Example: Branches step-by-step

eval ite_false:
ITE FALSE e1 e2
=\> (\ b x y -> b x y) FALSE e1 e2 -- expand def ITE
=\> (\ x y -> FALSE x y) e1 e2 -- beta-step
=\> (\ y -> FALSE e1 y) e2 -- beta-step
=\> FALSE e1 e2 -- expand def TRUE
=\> (\ x y -> y) e1 e2 -- beta-step
=\> (\ y -> y) e2 -- beta-step
=\> e2

Boolean operators

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

\[
\text{let } \text{NOT} = \ b \rightarrow ??? \\
\text{let } \text{AND} = \ b1 \ b2 \rightarrow ??? \\
\text{let } \text{OR} = \ b1 \ b2 \rightarrow ???
\]
Boolean operators

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

```haskell
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

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

such that

```plaintext
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 1st value
let SND = \p -> p FALSE -- call w/ FALSE, get 2nd value
```

Exercise: Triples?

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

```plaintext
let TRIPLE = \x y z -> ???
let FST3 = \t -> ???
let SND3 = \t -> ???
let TRD3 = \t -> ???
```
Exercise: Triples?

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

```latex
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 \(\lambda\)-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.

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

Pairs: Implementation

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

let ONE = λx → x
let TWO = λx → f (f x)
let THREE = λx → f (f (f x))
let FOUR = λx → f (f (f (f x)))
let FIVE = λx → f (f (f (f (f x))))
let SIX = λx → f (f (f (f (f (f x)))))
...

λ-calculus: Increment

-- Call 'f' on 'x' one more time than 'n' does
let INC = λn → (λx → ???)

• Example

eval inc_zero :
  INC ZERO
  => (λn x → f (n f x)) ZERO
  => (λx → f (ZERO f x))
  => (λx → f x)
  => ONE
λ-calculus: Addition

```haskell
let ADD = \n m -> n INC m

• Example

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

λ-calculus: Multiplication

```haskell
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 (structs, tuples) [done]
  - Numbers [done]
  - Functions [we got those]
  - Recursion
\(\lambda\)-calculus: Recursion

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

\(\lambda\)-calculus: Recursion

- No! Named terms in Elsa are just syntactic sugar
- To translate an Elsa term to \(\lambda\)-calculus: replace each name with its definition
  \[
  \begin{align*}
  \lambda n \rightarrow & \ \text{ITE (ISZ } n) \\
  & \ \text{ZERO} \\
  & \ \text{(ADD } n \ (\text{SUM (DEC } n))) \quad -- \ \text{But SUM is} \\
  & \ \text{not a thing!}
  \end{align*}
  \]
- **Recursion**: Inside this function I want to call the same function on \(\text{DEC } n\)
- 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 \(\text{DEC } n\)
  - Inside this function I want to call a function on \(\text{DEC } n\)
  - And BTW, I want it to be the same function
- **Step 1**: Pass in the function to call “recursively”

```lambda
let \text{STEP} =
\begin{align*}
\lambda \text{rec} \rightarrow \\
\lambda n \rightarrow & \ \text{ITE (ISZ } n) \\
& \ \text{ZERO} \\
& \ \text{(ADD } n \ (\text{rec (DEC } n))) \quad -- \ \text{Call some rec}
\end{align*}
```

-
**λ-calculus: Recursion**

- 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)))
  ```

**λ-calculus: Fixpoint Combinator**

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

  ```
  FIX STEP
  => STEP (FIX STEP)
  ```

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

- Once we have it, we can define:

  ```
  let SUM = FIX STEP
  ```

- Then by property of `FIX` we have:

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

**eval sum_one:**

```
**λ-calculus: Fixpoint Combinator**

- So how do we define FIX?

- Remember Ω? It *replicates itself!*
  \[(\lambda x \rightarrow x \ x) \ (\lambda x \rightarrow x \ x)\]
  \[=_{\beta} (\lambda x \rightarrow x \ x) \ (\lambda x \rightarrow x \ x)\]

- We need something similar but more involved.

---

**λ-calculus: Fixpoint Combinator**

- The Y combinator discovered by Haskell Curry:

  ```
  let FIX = \stp \rightarrow (\lambda x \rightarrow \stp (x \ x)) (\lambda x \rightarrow \stp (x \ x))
  ```

- How does it work?

  ```
  eval fix_step:
  FIX STEP
  =_{\beta} (\lambda x \rightarrow \stp (x \ x)) (\lambda x \rightarrow \stp (x \ x)) STEP
  =_{\beta} (\lambda x \rightarrow STEP (x \ x)) (\lambda x \rightarrow STEP (x \ x))
  =_{\beta} STEP (((\lambda x \rightarrow STEP (x \ x)) (\lambda x \rightarrow STEP (x \ x)))
  ```

  `^^^^^^^^^^ this is FIX STEP ^^^^^^^^^^^`

---

**Programming in λ-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