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

• 1936: Formalization
  - Alan Turing: Turing machines
  - Alonzo Church: lambda calculus

  \[
  e ::= x \\
  \quad | \quad \lambda x \to e \\
  \quad | \quad e1 e2
  \]
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")
\(x \to x\)

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

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

QUIZ: Lambda syntax
Which of the following terms are syntactically incorrect? *

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

http://tiny.cc/cmps112-lambda-grp
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>$\lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow e))$</td>
<td>$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$</td>
</tr>
<tr>
<td>$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$</td>
<td>$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$</td>
</tr>
<tr>
<td>$((\lambda e1 \ e2) \ e3) \ e4$</td>
<td>$\lambda e1 \ e2 \ e3 \ e4$</td>
</tr>
</tbody>
</table>

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

$((\lambda y \rightarrow y) \ \text{apple} \ \text{banana})$ -- *applied to two arguments*

Semantics: What programs mean

- How do I “run” or “execute” a $\lambda$-term?
- Think of middle-school algebra:
  
  > *Simplify expression:*
  
  $(x + 2)*((3*x - 1)$

  
  $=$

  $???

  
  *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:
  \[
  x \; y \\
  \lambda y \to x \; y \\
  (\lambda x \to \lambda y \to y) \; x \\
  \lambda x \to (\lambda y \to y) \; x
  \]
  - \( x \) is outside the scope
  - of the \( \lambda x \) binder;
  - 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:

\[
FV(x) = \{x\}
\]
\[
FV(x \rightarrow e) = FV(e) \setminus \{x\}
\]
\[
FV(e1 \ e2) = FV(e1) \cup FV(e2)
\]
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

$$\left( \lambda x \rightarrow e_1 \right) e_2 \rightarrow_e e_1[x := e_2]$$

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 \(\left( \lambda x \rightarrow e_1 \right) e_2\) \(\beta\)-steps to \(e_1[x := e_2]\)

Examples

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

Is this right? Ask Elsa!

\(\left( \lambda f \rightarrow f \left( \lambda x \rightarrow x \right) \right) \left( \text{give apple} \right) = \text{b} \rightarrow \text{???}\)

Examples

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

Is this right? Ask Elsa!

\(\left( \lambda f \rightarrow f \left( \lambda x \rightarrow x \right) \right) \left( \text{give apple} \right) = \text{b} \rightarrow \text{give apple} \left( \lambda x \rightarrow x \right)\)
QUIZ: B-Reduction 1

\((\lambda x \to (\lambda y \to y))\) apple = b> ??? *

A. apple
B. \(\lambda y \to\) apple
C. \(\lambda x \to\) apple
D. \(\lambda y \to\) y
E. \(\lambda x \to\) y

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

QUIZ: B-Reduction 1

\((\lambda x \to (\lambda y \to y))\) apple = b> ??? *

A. apple
B. \(\lambda y \to\) apple
C. \(\lambda x \to\) apple
D. \(\lambda y \to\) y
E. \(\lambda x \to\) y

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

QUIZ: B-Reduction 2

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

A. apple (\(\lambda x \to\) x)
B. apple (\(\lambda\) apple \(\to\) apple)
C. apple (\(\lambda x \to\) apple)
D. apple
E. \(\lambda x \to\) x

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

\((x \to x (x \to x))\) apple \(\Rightarrow\) ???

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

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

A Tricky One

\((x \to (y \to x))\) y
\(\Rightarrow\) \(\text{\textbackslash}y \to y\)

Is this right?

**Problem:** the free y in the argument has been *captured* by \(\text{\textbackslash}y\)!

**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:

\[(\Lambda x \to e_1) e_2 \Rightarrow e_1[x := e_2]\]

where \(e_1[x := e_2]\) means “*e1 with all free occurrences of x replaced with e2*”

- \(e_1\) 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:

\[ x[x := e] = e \]
\[ y[x := e] = y \quad \text{-- assuming } x \neq y \]
\[ (e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e]) \]
\[ \lambda x \rightarrow e_1[x := e] = \lambda x \rightarrow e_1 \quad \text{-- why just } e_1? \]

\[ \lambda y \rightarrow e_1[x := e] \]
\[ \mid \text{ not } (y \in \text{FV}(e)) = \lambda y \rightarrow e_1[x := e] \]
\[ \mid \text{ otherwise } = \text{undefined -- but what then???} \]

 Rewrite rules of lambda calculus

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

Semantics: \( \alpha \)-Reduction

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

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

\[ x \rightarrow e \quad \Rightarrow \quad y \rightarrow e[x := y] \]

\textbf{where} not (y in FV(e))

- Example:

\[ \begin{align*}
  x \rightarrow x & \Rightarrow a \\
  y \rightarrow y & \Rightarrow a \\
  z \rightarrow z & 
\end{align*} \]

- All these expressions are α-equivalent

Example

What’s wrong with these?

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

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

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

The Tricky One

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

\[ \Rightarrow \text{??} \]

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

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

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

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

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. \(x \, (y \rightarrow y)\)
- 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 \text{ evaluates to } e' \text{ if }
  1. There is a sequence of stops
     \[ e \Rightarrow e_1 \Rightarrow \ldots \Rightarrow e_N \Rightarrow e' \]
     where each \Rightarrow is either \Rightarrow a or \Rightarrow b and \( N \geq 0 \)
  2. \( e' \) is in normal form

Example of evaluation

\[
\begin{align*}
(\lambda x \to x) \text{ apple} \\
&\Rightarrow b> \text{ apple}
\end{align*}
\]

\[
\begin{align*}
(\lambda f \to f (\lambda x \to x)) (\lambda x \to x) \\
&\Rightarrow ???
\end{align*}
\]

\[
\begin{align*}
(\lambda x \to x) \text{ (} \lambda x \to x) \\
&\Rightarrow ???
\end{align*}
\]
Example of evaluation

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

*\text{b}*\ text{apple}

\(((\lambda \to \times) f (\lambda \to \times)) (\times \to \times)\)

*\text{b}*\ (\times \to \times) (\times \to \times)

*\text{b}*\ \times \to \times

\((\lambda \to \times \times) (\times \to \times)\)

*\text{b}*? ??

Example of evaluation

\((\times \to \times) \text{ apple} \)

*\text{b}*\ text{apple}

\(((\times \to \times) f (\times \to \times)) (\times \to \times)\)

*\text{b}*\ (\times \to \times) (\times \to \times)

*\text{b}*\ \times \to \times

\((\lambda \to \times \times) (\times \to \times)\)

*\text{b}*\ (\times \to \times) (\times \to \times)

*\text{b}*\ \times \to \times

Elsa shortcuts

- **Named \(\lambda\)-terms**

  ```
  \text{let ID} = \times \to \times \ -- \ \text{abbreviation for} \ \times \to \times
  ```

- To substitute a name with its definition, use a \(\text{=d}\) step:

  ```
  ID \text{ apple}
  =d> (\times \to \times) \text{ apple} \ -- \ expand \ \text{definition}
  =b> \text{ apple} \ -- \ 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 $a\rightarrow$, $b\rightarrow$, or $d\rightarrow$
  - $e_1 \rightsquigarrow e_2$: $e_1$ evaluates to $e_2$
- What is the difference?

Non-Terminating Evaluation

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

- 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 = (\lambda x \to x \ x) \ (\lambda x \to x \ x)
  (\lambda x \to \ y \to 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
  
  \[
  \begin{align*}
  \text{let } \text{TRUE} & = ??? \\
  \text{let } \text{FALSE} & = ??? \\
  \text{let } \text{ITE} & = \lambda b x y . ??? \quad \text{-- if } b \text{ then } x \text{ else } y
  \end{align*}
  \]

  such that

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

  (Here, \text{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

eval ITE:
ITE TRUE e1 e2
  = e (\b x y -> b x y) TRUE e1 e2 -- expand def ITE
  = e (\x y -> TRUE x y) e1 e2 -- beta-step
  = e (\y -> TRUE e1 y) e2 -- beta-step
  = e (TRUE e1 e2) -- expand def TRUE
  = e (\x y -> x) e1 e2 -- beta-step
  = e (\y -> e1) e2 -- beta-step
  = e 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:
ITE FALSE e1 e2
  -- fill the steps in!
  = e e2
Example: Branches step-by-step

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

\texttt{let NOT = \ b \rightarrow ???}
\texttt{let AND = \ b1 b2 \rightarrow ???}
\texttt{let OR = \ b1 b2 \rightarrow ???}

Boolean operators

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

\texttt{let NOT = \ b \rightarrow ITE b FALSE TRUE}
\texttt{let AND = \ b1 b2 \rightarrow ITE b1 b2 FALSE}
\texttt{let OR = \ b1 b2 \rightarrow ITE b1 TRUE b2}
Boolean operators

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

```ml
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
    { 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 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?

```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 \(\lambda\)-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 \(0, 1, 2, \ldots\)
- 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.

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

Pairs: Implementation

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

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

QUIZ: Church Numerals

Which of these is a valid encoding of ZERO? *

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

http://tiny.cc/cmps112-church-ind
**QUIZ: Church Numerals**

Which of these is a valid encoding of ZERO? *

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

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

---

**λ-calculus: Increment**

```latex
\text{-- Call \`f\' on \`x\' one more time than \`n\' does}
\text{let INC = \( \lambda n . (\lambda f f . f (n f x)) \)}
```

- Example

```plaintext
eval inc_zero :
    INC ZERO
  =\> (\lambda f f . f (n f x)) ZERO
  =\> \lambda f x . f (ZERO f x)
  =\> \lambda f x . f x
  =\> ONE
```

---

**QUIZ: ADD**

How shall we implement ADD? *

- A: let ADD = \( \lambda n m . n \) INC m
- B: let ADD = \( \lambda n m . INC n m \)
- C: let ADD = \( \lambda n m . n \) INC m
- D: let ADD = \( \lambda n m . (n m \) INC
- E: let ADD = \( \lambda n m . n (INC m) \)

http://tiny.cc/cmps112-add-ind
QUIZ: ADD

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)

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

λ-calculus: Addition

-- Call \f\n on \x\n exactly \n + m\n times
let ADD = \n m -> n INC m

• Example

eval add_one_zero :
ADD ONE ZERO
~~> ONE

QUIZ: MULT

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)

http://tiny.cc/cmps112-mult-ind
QUIZ: MULT

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

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

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 (structs, tuples) [done]
  - Numbers [done]
  - Functions [we got those]
  - Recursion
λ-calculus: Recursion

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

\[
\lambda n \rightarrow \ldots \quad = 1 + 2 + \ldots + n
\]

QUIZ: SUM

Is this a correct implementation of SUM? *

\[
\text{let } \text{SUM} = \lambda n \rightarrow \text{ITE ISZ n ZERO (ADD n (SUM (DEC n)))}
\]

- A. Yes
- B. No

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

QUIZ: SUM

Is this a correct implementation of SUM? *

\[
\text{let } \text{SUM} = \lambda n \rightarrow \text{ITE ISZ n ZERO (ADD n (SUM (DEC n)))}
\]

- A. Yes
- B. No

http://tiny.cc/cmps112-sum-grp
**λ-calculus: Recursion**

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

---

**λ-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”

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

---

**λ-calculus: Recursion**

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

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

- **Step 2**: Do something clever to \text{STEP}, so that the function passed as \text{rec} itself becomes

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

**λ-calculus: Fixpoint Combinator**

eval sum_one:

```plaintext
sum_one = STEP SUM ZERO -- (1)
((\rec n -> ITE (ISZ n) ZERO (ADD n (rec (DEC n)))) SUM ONE)

(FIX STEP)

-- the magic happened!

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

-- def of ISZ, ITE, DEC, ...

ADD ZERO ((\n -> ITE (ISZ n) ZERO (ADD n (SUM (DEC n)))) ONE)
```

**λ-calculus: Fixpoint Combinator**

- So how do we define `FIX`?

- Remember \( \Omega \)? It *replicates itself!*

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

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

  We need something similar but more involved.
**λ-calculus: Fixpoint Combinator**

- The Y combinator discovered by Haskell Curry:
  
  ```
  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 ^^^^^^^^^^'
  ```

**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**