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:

\[
\begin{array}{c}
32 \\
32 \\
231 \\
224 \\
72 \\
64 \\
8 \\
\end{array}
\]

can be computed by a human with pen and paper, following an algorithm

1936: Formalization

What is the smallest universal language?

Alan Turing
The Next 700 Languages

1930s

Alonzo Church

John McCarthy
LISP
1950s
SAIL
Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus.

Peter Landin, 1966

The Lambda Calculus

Has one feature:

- Functions
No, really

- Assignment \( x = x + 1 \)
- 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)
  - $(\! \neg x \rightarrow e)$
  - $x$ is the *formal* parameter, $e$ is the *body*
  - “for any $x$ compute $e$”

- **Application** (aka function call)

$$
E ::= x, y, z, \ldots
| (\! \neg x \rightarrow e)
| (e e)
| \text{func}(x)\{ E \}
| E (E)
$$
○ (e1 e2)
○ e1 is the function, e2 is the argument
○ in your favorite language: e1(e2)

(Here each of e, e1, e2 can itself be a variable, abstraction, or application)

Examples

\(\text{x} \rightarrow \text{x}\)  -- The "identity function" (id)
-- ("for any x compute x")

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

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

QUIZ

Which of the following terms are syntactically incorrect?

A. \((\text{x} \rightarrow \text{x}) \rightarrow \text{y}\)  -- NOT valid LC expression

\(\text{e} \rightarrow \text{e}^{\text{not}}\) "function that takes (x->x) as input"
Examples

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

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

\( \lambda f . f (\lambda x . x) \)  -- 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!

How do I apply a function to two arguments?

- e.g. apply \( x \rightarrow (y \rightarrow y) \) to apple and banana?

\(((x \rightarrow (y \rightarrow \ldots)) \text{ apple}) \text{ banana})\)

\( (\ x \ y \rightarrow y) \text{ apple} \text{ banana} \)

\(((\ x \rightarrow (y \rightarrow y)) \text{ apple}) \text{ banana}) \text{ -- first apply to apple,}
   \text{ -- then apply the result to banana} \)
**Syntactic Sugar**

\[
((x \ y) \ z) \quad x \ y \ z
\]

instead of

we write

<table>
<thead>
<tr>
<th>\x -&gt; (\y -&gt; (\z -&gt; e))</th>
<th>\x -&gt; \y -&gt; \z -&gt; e</th>
</tr>
</thead>
<tbody>
<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 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 \(\lambda\)-term?
Think of middle-school algebra:

--- Simplify expression:

\[
(1 + 2) \times ((3 \times 8) - 2)
\]

\[
= 3 \times (24 - 2)
\]

\[
= 3 \times 22
\]

\[
= 66
\]

Execute = rewrite step-by-step

- Following simple rules
- until no more rules apply
Rewrite Rules of Lambda Calculus

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

But first we have to talk about scope

"The scope of a variable is the part/region of the code where you can access that variable."

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:

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

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

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

\textbf{QUIZ}

In the expression $(\x \to x) \ x$, is $x$ bound or free?

A. first occurrence is bound, second is bound $x$

B. first occurrence is bound, second is free \checkmark

C. first occurrence is free, second is bound $\gamma$

D. first occurrence is free, second is free $\times$
EXERCISE: Free Variables

An variable \( x \) is free in \( e \) if there exists a free occurrence of \( x \) in \( e \).

We can formally define the set of all free variables in a term like so:

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(\lambda x \to e) &= FV(e) - \{x\} \\
FV(e1 e2) &= FV(e1) \cup FV(e2)
\end{align*}
\]

\[
\begin{align*}
FV(\lambda x \to x) &= \emptyset \\
FV(\lambda x \to y) &= \{y\} \\
FV((\lambda x \to x) x) &= \{x\} \\
FV(x y) &= \{x, y\} \\
FV(x) &= \{x\}
\end{align*}
\]

Closed Expressions

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

- Closed expressions are also called combinators.

\[
\text{Y - COMBINATOR "STARTUP INCUBATOR"}
\]

What is the shortest closed expression?

\( \lambda x \to x \)
Rewrite Rules of Lambda Calculus

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

Semantics: Redex

A redex is a term of the form

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

A function $(\lambda x \to e_1)$

- $x$ is the parameter
- $e_1$ is the returned expression

Applied to an argument $e_2$

- $e_2$ is the argument
Semantics: $\beta$-Reduction

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

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

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 $e_1[x := e_2]$

Redex Examples
(\x \to \ x) \text{apple} \\
=\text{b} \to \text{apple}

Is this right? Ask Elsa (https://elsa.goto.ucsd.edu/index.html#?demo=permalink%2F1695925711_23.lc)

\textbf{QUIZ}

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

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

\((\langle x \rightarrow (\langle x \rightarrow x) \rangle \rangle \text{apple}) = b \Rightarrow \text{???}\)

A. apple apple apple apple
B. y apple y apple
C. y y y y
D. apple

QUIZ

\((\langle x \rightarrow x \rangle \text{apple}) = b \Rightarrow \text{???}\)

A. apple (\(\langle x \rightarrow x \rangle\))
B. apple (\(\langle \text{apple} \rightarrow \text{apple} \rangle\))
C. apple (\(\langle x \rightarrow \text{apple} \rangle\))
D. apple
E. \(\langle x \rightarrow x \rangle\)
**EXERCISE**

What is a $\lambda$-term `fill_this_in` such that

```
fill_this_in apple =b> banana
```

ELSA: https://elsa.goto.ucsd.edu/index.html

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

---

**A Tricky One**
Something is Fishy

\( (\lambda x \rightarrow (\lambda y \rightarrow x)) \, y \)
\[ =_{b} \, \lambda y \rightarrow y \]

Is this right?

**Problem:** The free `y` in the argument has been captured by `\y` in body!

**Solution:** Ensure that formals in the body are different from free-variables of argument!
Capture-Avoiding Substitution

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

\[
(x \to e1) e2 \rightarrow e1[x := e2]
\]

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

Formally:

\[
\begin{align*}
\text{cat}[\text{cat} := \text{horse}] & \rightarrow \text{horse} \\
x[x := e] & = e \\
\text{cat}[\text{dog} := \text{horse}] & \rightarrow \text{cat} \\
y[x := e] & = y \quad \text{-- as } x \neq y \\
(e1 e2)[x := e] & = (e1[x := e]) (e2[x := e]) \\
(x \to e1)[x := e] & = x \to e1 \quad \text{-- Q: Why `e1` unchanged?} \\
(y \to e1)[x := e] & = y \to e1[x := e] \\
\mid \text{not } (y \in \text{FV}(e)) & = y \to e1[x := e]
\end{align*}
\]

Oops, but what to do if \( y \) is in the free-variables of \( e \)?

- i.e. if \( \text{\textbackslash y} \rightarrow \ldots \) may capture those free variables?

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

Semantics: $\alpha$-Renaming

\[ x \to e \Rightarrow y \to e[x := y] \]

where not (y in FV(e))

- We rename a formal parameter $x$ to $y$
- By replace all occurrences of $x$ in the body with $y$
- We say that $x \to e \alpha$-steps to $y \to e[x := y]$

Example:

\[ x \to x \Rightarrow y \to y \Rightarrow z \to z \]

All these expressions are $\alpha$-equivalent

What’s wrong with these?

- (A)

\[ f \to f x \Rightarrow x \to x x \]
Tricky Example Revisited

\[ (\lambda x \rightarrow (\lambda y \rightarrow y)) \ y \ =_{\alpha} \ (\lambda x \rightarrow (\lambda z \rightarrow z)) \ z \]

\[ (\lambda x \rightarrow (\lambda y \rightarrow x)) \ y \quad \text{-- rename 'y' to 'z' to avoid capture} \]
\[ =_{\alpha} (\lambda x \rightarrow (\lambda z \rightarrow x)) \ y \quad \text{-- now do b-step without capture!} \]
\[ =_{b} \ (\lambda z \rightarrow y) \]

To avoid getting confused,

- you can **always rename** formals,
- so different **variables** have different **names**!
Normal Forms

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

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

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

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

E. C and D
Semantics: Evaluation

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

1. There is a sequence of steps

$$e =?\ e_1 =?\ ... =?\ e_N =?\ e'$$

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

2. $e'$ is in normal form

Examples of Evaluation

$$(\lambda x \to x)\ apple$$

$=b>\ apple$

$$(\lambda f \to f (\lambda x \to x))\ (\lambda x \to x)$$

$$=?>\ ???$$

$$=?\ \ ???$$

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

$$=?>\ ???$$
**Elsa shortcuts**

Named $\lambda$-terms:

$$\textbf{let} \ ID = \lambda x \to x \quad -- \textit{abbreviation for} \ \lambda x \to x$$

To substitute name with its definition, use a $=d>$ step:

$$\text{ID apple}$$

$$=d> (\lambda x \to x) \text{ apple} \quad -- \textit{expand definition}$$

$$=b> \text{ apple} \quad -- \textit{beta-reduce}$$

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 $e2$ and $e2$ is in normal form

**EXERCISE**

Fill in the definitions of FIRST, SECOND and THIRD such that you get the following behavior in elsa
let FIRST = fill_this_in
let SECOND = fill_this_in
let THIRD = fill_this_in

let OMEGA = (\x -> x) (\x -> x)

Non-Terminating Evaluation

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

Some programs 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 -> 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), lists, trees,...
- 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 $b$?

$\text{ITE} = \lambda b \ x \ y \rightarrow (b \ x \ y)$

$\text{true} = \lambda e \ f \rightarrow e$

$\text{false} = \lambda e \ f \rightarrow f$

TRUE $\equiv (\lambda x \ (y \rightarrow x))$

FALSE $\equiv (\lambda x \ (y \rightarrow y))$

TRUE $\times$ y $\equiv x$

TRUE $\equiv \lambda y \rightarrow x$

TRUE $\equiv (\lambda x \rightarrow y \rightarrow x)$

FALSE $\times$ y $\equiv y$

FALSE $\equiv (\lambda x \rightarrow y)$

TRUE $\times$ y $\equiv x$

FALSE $\times$ y $\equiv y$

Foo $\equiv (\lambda x \ e)$

$\text{FOO} = \lambda x \rightarrow e$
Make a binary choice

- if b then e1 else e2

**Booleans: API**

We need to define three functions

```plaintext
let TRUE = ???
let FALSE = ???
let ITE = \b x y -> ??? -- if b then x else y
```

such that

- ITE TRUE apple banana => apple
- ITE FALSE apple banana => banana

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

\begin{verbatim}
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)
\end{verbatim}

Example: Branches step-by-step

eval ite_true:

\begin{verbatim}
ITE TRUE e1 e2
\end{verbatim}
**Example: Branches step-by-step**

Now you try it!

Can you fill in the blanks to make it happen? (https://elsa.goto.ucsd.edu/index.html#?demo=ite.lc)

```plaintext
eval ite_false:
   ITE FALSE e1 e2

  -- fill the steps in!

  =b> e2
```

---

**EXERCISE: Boolean Operators**

ELSA: https://goto.ucsd.edu/elsa/index.html Click here to try this exercise
(https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585435168_24442.lc)

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

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

When you are done, you should get the following behavior:
eval ex_not_t:
    NOT TRUE => FALSE

eval ex_not_f:
    NOT FALSE => TRUE

eval ex_or_ff:
    OR FALSE FALSE => FALSE

eval ex_or_ft:
    OR FALSE TRUE => TRUE

eval ex_or_tf:
    OR TRUE FALSE => TRUE

eval ex_or_tt:
    OR TRUE TRUE => TRUE

eval ex_and_ff:
    AND FALSE FALSE => FALSE

eval ex_and_ft:
    AND FALSE TRUE => FALSE

eval ex_and_tf:
    AND TRUE FALSE => FALSE

eval ex_and_tt:
    AND TRUE TRUE => TRUE

Programming in λ-calculus
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

\begin{verbatim}
let PAIR = \x y -> ???   -- Make a pair with elements x and y
                          -- \{ \text{fst} : x, \text{snd} : y \}
let FST   = \p -> ???   -- Return first element
                      -- \p.fst
let SND   = \p -> ???   -- Return second element
                      -- \p.snd
\end{verbatim}

such that

\begin{verbatim}
eval ex_fst:  
    FST (PAIR apple banana) =*\rightarrow apple
\end{verbatim}

\begin{verbatim}
eval ex_snd:  
    SND (PAIR apple banana) =*\rightarrow banana
\end{verbatim}

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 \( y \))

\begin{verbatim}
let PAIR = \x y -> (\b -> ITE b x y)
let FST  = \p -> p \text{TRUE}    -- call w/ \text{TRUE}, get first value
let SND  = \p -> p \text{FALSE}   -- call w/ \text{FALSE}, get second value
\end{verbatim}
**EXERCISE: Triples**

How can we implement a record that contains three values?

ELSA: https://goto.ucsd.edu/elsa/index.html

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?
demo=permalink%2F1585434814_24436.lc)

```ml
let TRIPLE = \x y z -> ???
let FST3 = \t -> ???
let SND3 = \t -> ???
let THD3 = \t -> ???
```

**eval ex1:**
```
FST3 (TRIPLE apple banana orange)
=> apple
```

**eval ex2:**
```
SND3 (TRIPLE apple banana orange)
=> banana
```

**eval ex3:**
```
THD3 (TRIPLE apple banana orange)
=> orange
```
Programming in $\lambda$-calculus

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

$\lambda$-calculus: Numbers

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

What do we do with natural numbers?

- Count: $\emptyset$, inc
- Arithmetic: dec, +, -, *
- Comparisons: $\equiv$, $\leq$, 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.

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

**Natural Numbers: Implementation**

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

\[
\begin{align*}
\text{let ONE} & = \lambda f \ x \to f \ x \\
\text{let TWO} & = \lambda f \ x \to f \ (f \ x) \\
\text{let THREE} & = \lambda f \ x \to f \ (f \ (f \ x)) \\
\text{let FOUR} & = \lambda f \ x \to f \ (f \ (f \ (f \ x))) \\
\text{let FIVE} & = \lambda f \ x \to f \ (f \ (f \ (f \ (f \ x)))) \\
\text{let SIX} & = \lambda f \ x \to f \ (f \ (f \ (f \ (f \ (f \ x))))) \\
\ldots
\end{align*}
\]
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 = \( f \ x \rightarrow f \ x \) \( \text{ONE!} \)
- D: let ZERO = \( x \rightarrow x \) \( \times \)
- E: None of the above

Does this function look familiar?

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

\[
\text{INC } n = \begin{align*}
&= \begin{cases}
  n f x & \quad \text{(}\ldots \quad f(f(f(x)))))\\
  & \quad \text{(}n f x)\text{ \ times}
\end{cases} \\
  \quad \text{=} \begin{cases}
  f (n f x) & \quad \text{\ times} \\
  f x & \quad \text{\ times}
\end{cases} \\
  \quad \text{=} \begin{cases}
  f (n f) & (x) \\
  f x & (n f)
\end{cases}
\]
\]

Example:

\[
\text{eval inc_zero :} \\
\quad \text{INC ZERO} \\
  =d> (\begin{cases}
  n f x & \rightarrow f (n f x)) \quad \text{ZERO} \\
  \end{cases} \\
  =b> \begin{cases}
  f x & \rightarrow f (ZERO f) \\
  \end{cases} \\
  =*> \begin{cases}
  f x & \rightarrow f x \\
  \end{cases} \\
  =d> \quad \text{ONE}
\]

EXERCISE

Fill in the implementation of ADD so that you get the following behavior

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585436042_24449.lc)
let ZERO = \x -> x
let ONE = \x -> f x
let TWO = \x -> f (f x)
let INC = \n f x -> f (n f x)

let ADD = fill_this_in

eval add_zero_zero:
  ADD ZERO ZERO =~~> ZERO

eval add_zero_one:
  ADD ZERO ONE =~~> ONE

eval add_zero_two:
  ADD ZERO TWO =~~> TWO

eval add_one_zero:
  ADD ONE ZERO =~~> ONE

eval add_one_zero:
  ADD ONE ONE =~~> TWO

eval add_two_zero:
  ADD TWO ZERO =~~> TWO

**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)
\(\lambda\)-calculus: Addition

\[
\text{let } \text{ADD} = \ \lambda \ n \ m \to \ n \ \text{INC} \ m
\]

Example:

\[
\text{eval add_one_zero} : \\
\text{ADD} \ \text{ONE} \ \text{ZERO} \\
\Rightarrow \text{ONE}
\]

**QUIZ**

How shall we implement MULT?

A. let MULT = \(\lambda \ n \ m \to n \ \text{ADD} \ m\)

B. let MULT = \(\lambda \ n \ m \to n \ (\text{ADD} \ m) \ \text{ZERO}\)

C. let MULT = \(\lambda \ n \ m \to m \ (\text{ADD} \ n) \ \text{ZERO}\)

D. let MULT = \(\lambda \ n \ m \to n \ (\text{ADD} \ m \ \text{ZERO})\)

E. let MULT = \(\lambda \ n \ m \to (n \ \text{ADD} \ m) \ \text{ZERO}\)
\( \lambda \text{-calculus: Multiplication} \)

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

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

Example:

\[
\text{eval two\_times\_three :} \\
\text{MULT two one} \\
\rightarrow \text{TWO}
\]

\[
\begin{align*}
\text{IS\_ZERO} & \rightarrow \text{TRUE} \\
\text{IS\_ZERO} \ \text{ONE} & \rightarrow \text{FALSE} \\
\text{IS\_ZERO} \ \text{TWO} & \rightarrow \text{FALSE}
\end{align*}
\]

\text{Programming in } \lambda \text{-calculus}
**λ-calculus: Lists**

Let's define an API to build lists in the λ-calculus.

**An Empty List**

NIL

**Constructing a list**

A list with 4 elements

```
CONS apple (CONS banana (CONS cantaloupe (CONS dragon NIL)))
```

Intuitively, CONS h t creates a new list with

- head h
- tail t

**Destructing a list**

- HEAD l returns the first element of the list
- TAIL l returns the rest of the list
\lambda\text{-calculus: Lists}

\begin{align*}
\text{let } &\text{NIL } = \text{???} \\
\text{let } &\text{CONS } = \text{???} \\
\text{let } &\text{HEAD } = \text{???} \\
\text{let } &\text{TAIL } = \text{???} \\
\text{eval } &\text{exHd:} \\
&\text{HEAD }\left(\text{CONS}\ \text{apple}\ \text{CONS}\ \text{banana}\ \text{CONS}\ \text{cantaloupe}\ \text{CONS}\ \text{dragon}\ \text{NIL}\right)) \\
&\Rightarrow\text{ apple} \\
\text{eval } &\text{exTl} \\
&\text{TAIL }\left(\text{CONS}\ \text{apple}\ \text{CONS}\ \text{banana}\ \text{CONS}\ \text{cantaloupe}\ \text{CONS}\ \text{dragon}\ \text{NIL}\right)) \\
&\Rightarrow\text{ CONS}\ \text{banana}\ \text{CONS}\ \text{cantaloupe}\ \text{CONS}\ \text{dragon}\ \text{NIL})
\end{align*}
EXERCISE: Nth

Write an implementation of \texttt{GetNth} such that

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

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

\[
\begin{align*}
\text{let } \text{GetNth} & = \text{??}
\end{align*}
\]

\[
\begin{align*}
\text{eval nth1} & : \quad \text{GetNth \ ZERO} (\text{CONS apple (CONS banana (CONS cantaloupe NIL))}) \\
& \Rightarrow \text{apple}
\end{align*}
\]

\[
\begin{align*}
\text{eval nth1} & : \quad \text{GetNth \ ONE} (\text{CONS apple (CONS banana (CONS cantaloupe NIL))}) \\
& \Rightarrow \text{banana}
\end{align*}
\]

\[
\begin{align*}
\text{eval nth2} & : \quad \text{GetNth \ TWO} (\text{CONS apple (CONS banana (CONS cantaloupe NIL))}) \\
& \Rightarrow \text{cantaloupe}
\end{align*}
\]

Click here to try this in elsa (https://goto.ucsd.edu/elsa/index.html#?
demo=permalink%2F1586466816_52273.lc)

\textit{λ-calculus: Recursion}

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

\[
\begin{align*}
\text{let } \text{SUM} & = \n \Rightarrow \ldots \ldots 0 + 1 + 2 + \ldots + n
\end{align*}
\]
such that we get the following behavior

eval exSum0: SUM ZERO  =~> ZERO 0

eval exSum1: SUM ONE  =~> ONE 0+1

eval exSum2: SUM TWO  =~> THREE 0+1+2

eval exSum3: SUM THREE =~> SIX 0+1+2+3

Can we write sum using Church Numerals?

Click here to try this in Elsa (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1586465192_52175.lc)

**QUIZ**

You can write \( SUM \) using numerals but its tedious.

Is this a correct implementation of \( SUM \)?

A. Yes

B. No
No!

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

\[
\lambda n \rightarrow \text{ITE} \ (\text{ISZ} \ n) \\
\quad \text{ZERO} \\
\quad (\text{ADD} \ n \ (\text{SUM} \ (\text{DEC} \ n))) \quad \text{-- But SUM is not yet defined!}
\]

Recursion:

- Inside this function
- Want to call the same function on DEC \( n \)

Looks like we can’t do recursion!

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

Right?

\(\lambda\)-calculus: Recursion

Think again!
Recursion:

Instead of

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

Let's try

- Inside this function I want to call some function rec on DEC n
- And BTW, I want rec to be the same function

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

```plaintext
let STEP = 
  \rec \rightarrow \n \rightarrow ITE (ISZ n)
      ZERO
    (ADD n (rec (DEC n))) -- Call some rec
```

Step 2: Do some magic to STEP, so rec is itself

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

That is, obtain a term MAGIC such that

```plaintext
MAGIC =*\rightarrow\ STEP MAGIC
```
λ-calculus: Fixpoint Combinator

Wanted: a λ-term FIX such that

- FIX STEP calls STEP with FIX STEP as the first argument:

\[(\text{FIX STEP}) =*\to \text{STEP (FIX STEP)}\]

\[\text{FIX STEP} =\text{STEP (FIX STEP)}\]

(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:

\[
\text{let SUM} = \text{FIX STEP}
\]

Then by property of FIX we have:

\[
\text{SUM} =*\to \text{FIX STEP} =*\to \text{STEP (FIX STEP)} =*\to \text{STEP SUM}
\]

and so now we compute:

eval sum_two:

\[
\begin{align*}
\text{SUM TWO} \\
=\to \text{STEP SUM TWO} \\
=\to \text{ITE (ISZ TWO) ZERO (ADD TWO (SUM (DEC TWO)))} \\
=\to \text{ADD TWO (SUM (DEC TWO))} \\
=\to \text{ADD TWO (SUM ONE)} \\
=\to \text{ADD TWO (STEP SUM ONE)} \\
=\to \text{ADD TWO (ITE (ISZ ONE) ZERO (ADD ONE (SUM (DEC ONE))))} \\
=\to \text{ADD TWO (ADD ONE (SUM (DEC ONE)))} \\
=\to \text{ADD TWO (ADD ONE (SUM ZERO))} \\
=\to \text{ADD TWO (ADD ONE (ITE (ISZ ZERO) ZERO (ADD ZERO (SUM DEC ZERO))))} \\
=\to \text{ADD TWO (ADD ONE (ZERO))} \\
=\to \text{THREE}
\end{align*}
\]

How should we define FIX??
The Y combinator

Remember $\Omega$?

\[
\lambda x. x \ x \ (\lambda x. x \ x)
\]

\[= b \ > \ (\lambda x. x \ x) \ (\lambda x. x \ x)\]

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

The Y combinator discovered by Haskell Curry:

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

How does it work?

eval fix_step:

\[
\begin{align*}
\text{FIX STEP} & \\
= d & > (\text{stp} \rightarrow (\lambda x. \text{stp} \ (x \ x)) \ (\lambda x. \text{stp} \ (x \ x))) \ \text{STEP} \\
= b & > (\lambda x. \text{STEP} \ (x \ x)) \ (\lambda x. \text{STEP} \ (x \ x)) \\
= b & > \text{STEP} \ ((\lambda x. \text{STEP} \ (x \ x)) \ (\lambda x. \text{STEP} \ (x \ x))) \\
\text{-- } & \text{ this is FIX STEP } \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow
\end{align*}
\]

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

Next week: We'll look at the language named after him (Haskell)