Lambda Calculus

Your Favorite Language

Probably has lots of features:

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

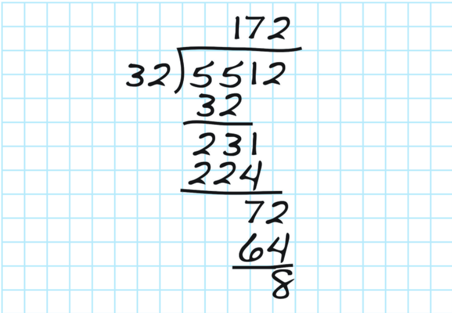
Which ones can we do without?

What is the **smallest universal language**?

What is computable?

Before 1930s

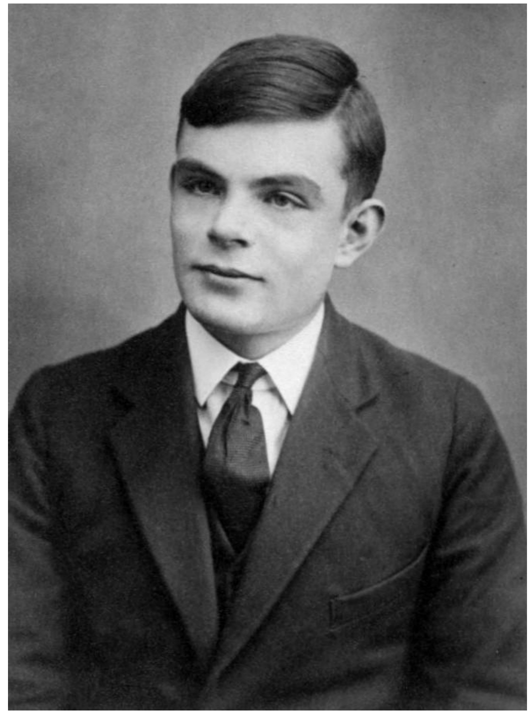
Informal notion of an **effectively calculable** function:



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

1936: Formalization

What is the **smallest universal language**?



Alan Turing

Alonzo Church

The Next 700 Languages

Peter Landin

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

Peter Landin, 1966

The Lambda Calculus

Has one feature:

- Functions

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

```
e ::= x
    | \x -> e
    | e1 e2
```

Programs are **expressions** `e` (also called **λ-terms**) of one of three kinds:

- **Variable**
  - `x`, `y`, `z`

- **Abstraction** (aka *nameless* function definition)
  - `\x -> e`
  - `x` is the *formal* parameter, `e` is the *body*
  - "for any `x` compute `e`"
- **Application** (aka function call)
  - `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

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

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

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

QUIZ

Which of the following terms are syntactically **incorrect**?

A. `\(\x -> x) -> y`

B. `\x -> x x`

C. `\x -> x (y x)`

D. A and C

E. all of the above

Examples

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

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

\f -> f (\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 -> (\y -> 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 -> (\y -> y)` to `apple` and `banana`?

```
(((\x -> (\y -> y)) apple) banana) -- first apply to apple,
                                   -- then apply the result to banana
```

Syntactic Sugar

instead of	we write
`\x -> (\y -> (\z -> e))`	`\x -> \y -> \z -> e`
`\x -> \y -> \z -> e`	`\x y z -> e`
`(((e1 e2) e3) e4)`	`e1 e2 e3 e4`

```
\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 λ-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. α-step (aka *renaming formals*)
2. β-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 `\x -> e`

- `x` is the newly introduced variable
- `e` is **the scope** of `x`
- any occurrence of `x` in `\x -> e` is **bound** (by the **binder** `\x`)

For example, `x` is bound in:

```
\x -> x
\x -> (\y -> 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                -- no binders at all!
\y -> x y          -- no \x binder
(\x -> \y -> y) x  -- x is outside the scope of the \x binder;
                   -- intuition: it's not "the same" x
```

QUIZ

In the expression `(\x -> 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

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)        = ???
FV(\x -> e) = ???
FV(e1 e2)   = ???
```

Closed Expressions

If `e` has *no free variables* it is said to be **closed**

- Closed expressions are also called **combinators**

What is the shortest closed expression?

Rewrite Rules of Lambda Calculus

1. α-step (aka *renaming formals*)
2. β-step (aka *function call*)

Semantics: β-Reduction

```
(\x -> e1) e2   =b>   e1[x := e2]
```

where `e1[x := e2]` means “ `e1` with all *free* occurrences of `x` replaced with `e2` ”

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 `(\x -> e1) e2` β-steps to `e1[x := e2]`

Examples

```
(\x -> x) apple
=b> apple
```

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

```
(\f -> f (\x -> x)) (give apple)
=b> ???
```

QUIZ

```
(\x -> (\y -> y)) apple
=b> ???
```

A. `apple`

B. `\y -> apple`

C. `\x -> apple`

D. `\y -> y`

E. `\x -> y`

QUIZ

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

A. `apple (\x -> x)`

B. `apple (\apple -> apple)`

C. `apple (\x -> apple)`

D. `apple`

E. `\x -> x`

A Tricky One

```
(\x -> (\y -> x)) y
=b> \y -> y
```

Is this right?

Something is Fishy

```
(\x -> (\y -> x)) y
=b> \y -> y
```

Is this right?

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

Solution: make sure that all *free variables* of the argument are different from the binders in the body.

Capture-Avoiding Substitution

We have to fix our definition of β-reduction:

```
(\x -> e1) e2   =b>   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

Formally:

```
x[x := e]            = e
y[x := e]            = y              -- assuming x /= y
(e1 e2)[x := e]      = (e1[x := e]) (e2[x := e])
(\x -> e1)[x := e]   = \x -> e1       -- why do we leave `e1` alone?
(\y -> e1)[x := e]
  | not (y in FV(e)) = \y -> e1[x := e]
  | otherise         = undefined      -- wait, but what do we do the
n???
```

Rewrite Rules of Lambda Calculus

1. α-step (aka *renaming formals*)
2. β-step (aka *function call*)

Semantics: α-Renaming

```
\x -> e   =a>   \y -> e[x := y]
  where not (y in FV(e))
```

- We can rename a formal parameter and replace all its occurrences in the body
- We say that `\x -> e` α-steps to `\y -> e[x := y]`

Example:

```
\x -> x   =a>   \y -> y   =a>    \z -> z
```

All these expressions are **α-equivalent**

What's wrong with these?

```
-- (A)
\f -> f x    =a>   \x -> x x

-- (B)
(\x -> \y -> y) y   =a>   (\x -> \z -> z) z

-- (C)
\x -> \y -> x y   =a>    \apple -> \orange -> apple orange
```

The Tricky One

```
(\x -> (\y -> x)) y
=a> ???
```

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

```
(\x -> e1) e2
```

A λ-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. `(\x -> x) y`

D. `x (\y -> y)`

E. C and D

Semantics: Evaluation

A λ-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 >= 0`

2. `e'` is in *normal form*

Examples of Evaluation

```
(\x -> x) apple
  =b> apple
```

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

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

Elsa shortcuts

Named λ-terms:

```
let ID = \x -> x  -- abbreviation for \x -> x
```

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

```
ID apple
  =d> (\x -> x x) apple  -- expand definition
  =b> apple              -- beta-reduce
```

Evaluation:

- `e1 =*> e2` : `e1` reduces to `e2` in 0 or more steps
    - where each step is `=a>` , `=b>` , or `=d>`
- `e1 =~> e2` : `e1` evaluates to `e2`

What is the difference?

Non-Terminating Evaluation

```
(\x -> x x) (\x -> x x)
  =b> (\x -> x x) (\x -> x x)
```

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

and never reduce to a normal form!

This combinator is called Ω

What if we pass Ω 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 λ-calculus

Real languages have lots of features

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

Lets see how to *encode* all of these features with the λ-calculus.

λ-calculus: Booleans

How can we encode Boolean values (`TRUE` and `FALSE`) as functions?

Well, what do we **do** with a Boolean `b` ?

Make a *binary choice*

- **`if`** `b` **`then`** `e1` **`else`** `e2`

Booleans: API

We need to define three functions

```
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

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

Example: Branches step-by-step

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

  -- fill the steps in!

  =b> e2
```

Boolean Operators

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

```
let NOT = \b     -> ???

let AND = \b1 b2 -> ???

let OR  = \b1 b2 -> ???
```

```
let NOT = \b     -> ITE b FALSE TRUE

let AND = \b1 b2 -> ITE b1 b2 FALSE

let OR  = \b1 b2 -> ITE b1 TRUE b2
```

Or, since `ITE` is redundant:

```
let NOT = \b     -> b FALSE TRUE

let AND = \b1 b2 -> b1 b2 FALSE

let OR  = \b1 b2 -> b1 TRUE b2
```

Which definition to do you prefer and why?

Programming in λ-calculus

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

λ-calculus: Records

Let's start with records with *two* fields (aka **pairs**)

What do we *do* with a pair?

1. **Pack two** items into a pair, then
2. **Get first** item, or
3. **Get second** item.

Pairs : API

We need to define three functions

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

such that

```
FST (PAIR apple banana) =~> apple
SND (PAIR apple banana) =~> banana
```

Pairs: Implementation

A pair of `x` and `y` is just something that lets you pick between `x` and `y`! (I.e. a function that takes a boolean and returns either `x` or `y`)

```
let PAIR = \x y -> (\b -> ITE b x y)
let FST  = \p -> p TRUE   -- call w/ TRUE, get first value
let SND  = \p -> p FALSE  -- call w/ FALSE, get second value
```

Exercise: Triples?

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

```
let TRIPLE = \x y z -> ???
let FST3  = \t -> ???
let SND3  = \t -> ???
let TRD3  = \t -> ???
```

Programming in λ-calculus

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

- **Functions** [we got those]
- Recursion

λ-calculus: Numbers

Let's start with **natural numbers** (0, 1, 2, …)

What do we *do* with natural numbers?

- Count: `0` , `inc`
- Arithmetic: `dec` , `+` , `-` , `*`
- Comparisons: `==` , `<=` , etc

Natural Numbers: API

We need to define:

- A family of **numerals**: `ZERO` , `ONE` , `TWO` , `THREE` , ...
- Arithmetic functions: `INC` , `DEC` , `ADD` , `SUB` , `MULT`
- Comparisons: `IS_ZERO` , `EQ`

Such that they respect all regular laws of arithmetic, e.g.

```
IS_ZERO ZERO       =~> TRUE
IS_ZERO (INC ZERO) =~> FALSE
INC ONE            =~> TWO
...
```

Natural Numbers: Implementation

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

```
let ONE   = \f x -> f x
let TWO   = \f x -> f (f x)
let THREE = \f x -> f (f (f x))
let FOUR  = \f x -> f (f (f (f x)))
let FIVE  = \f x -> f (f (f (f (f x))))
let SIX   = \f x -> f (f (f (f (f (f x)))))
...
```

QUIZ: Church Numerals

Which of these is a valid encoding of `ZERO` ?

- **A**: `let ZERO = \f x -> x`
- **B**: `let ZERO = \f x -> f`
- **C**: `let ZERO = \f x -> f x`
- **D**: `let ZERO = \x -> x`
- **E**: None of the above

Does this function look familiar?

λ-calculus: Increment

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

Example:

```
eval inc_zero :
  INC ZERO
  =d> (\n f x -> f (n f x)) ZERO
  =b> \f x -> f (ZERO f x)
  =*> \f x -> f x
  =d> ONE
```

QUIZ

How shall we implement `ADD`?

A. `let ADD = \n m -> n INC m`

B. `let ADD = \n m -> INC n m`

C. `let ADD = \n m -> n m INC`

D. `let ADD = \n m -> n (m INC)`

E. `let ADD = \n m -> n (INC m)`

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

Example:

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

QUIZ

How shall we implement `MULT`?

A. `let MULT = \n m -> n ADD m`

B. `let MULT = \n m -> n (ADD m) ZERO`

C. `let MULT = \n m -> m (ADD n) ZERO`

D. `let MULT = \n m -> n (ADD m ZERO)`

E. `let MULT = \n m -> (n ADD m) ZERO`

λ-calculus: Multiplication

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

Example:

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

Programming in λ-calculus

- **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`:

```
\n -> ...           -- 1 + 2 + ... + n
```

QUIZ

Is this a correct implementation of `SUM`?

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

A. Yes

B. No

No!

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

```
\n -> ITE (ISZ n)
        ZERO
        (ADD n (SUM (DEC n))) -- But SUM is 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"

```
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:
  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
```

How should we define `FIX` ???

The Y combinator

Remember Ω?

```
(\x -> x x) (\x -> x x)
=b> (\x -> x x) (\x -> x x)
```

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

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 ^^^^^^^^^^^
```

That’s all folks!
