# CSE114A Lecture 2

Agenda:

- intro to lambda calculus

the Greek letter $\lambda$ — a system of reasoning.

1936 – Turing machines were invented (Alan Turing)

1936 – $\lambda$-calculus was invented (Alonzo Church)

both universal models of computation.

What features would you expect a PL to have?

conditionals

variables

Functions

objects/classes

expressions

types (booleans, numeric types, user-defined types)

operators (+, -, etc.)

looping constructs (for, while, ...)

(Lambda)

$\lambda$-calculus only has functions.
everything else we might want has to be encoded using functions!

---

- functions
  - way to define them (function definition)
  - way to call them (function application) aka function call

$$e ::= \boxed{x, y, z} \quad \leftarrow \text{variables}$$

$$\mid \boxed{\lambda x.\ e} \quad \leftarrow \text{function definition.}$$

$$\mid \boxed{e_1 \quad e_2} \quad \leftarrow \text{function application}$$

expression

formal parameter — body

function we're calling — argument to that function

✓ Syntax - what programs look like.

Semantics - what programs mean.

$\lambda y.\ y$ ← also the identity function

$\boxed{\lambda x.\ x}$ ← the identity function.

takes an argument — and just returns that argument

```
def identity(y):
    return y
```
← identity function in Python.

$\lambda z.\ (\lambda x.\ x)$ ← a function that returns the identity function

$\lambda y.\ (\lambda x.\ x)$ ← so is this

$\lambda z.\ (\lambda y.\ y)$ ← and so is this

$\lambda f.\ f\ (\lambda x.\ x)$ ← a function that applies its argument to the identity function

an application!

| paper syntax | Elsa syntax |
|---|---|
| $\lambda x.\ x$ | `\x -> x` |
| $\lambda f.\ f\ (\lambda x.\ x)$ | `\f -> f (\x -> x)` |

What about semantics?
We've talked about syntax, but what do λ-calculus programs mean?

One way to talk about meaning is **operational semantics**:
how do programs execute step by step?

The essence of computation in λ-calculus is substitution.

```
def identity(x):
    return x
```

> identity(3)
3

We substituted 3 for x.

evaluates to

$$\underbrace{(\lambda x.\ e_1)}_{\text{function}}\ \underbrace{e_2}_{\text{argument}} \longrightarrow e_1[x := e_2]$$

"$e_1$, but with all occurrences of x in $e_1$ replaced by $e_2$."

$$\underbrace{(\lambda x.x)}_{\text{identity function}}\ \underbrace{(\lambda y.y)}_{\text{argument}} \longrightarrow x[x := \lambda y.y]$$
$$= \lambda y.y$$

$$\underbrace{(\lambda x.x)}_{\text{identity function}}\ \underbrace{y}_{\text{argument}} \qquad x[x := y]$$
$$y$$

---

Let's try encoding Booleans and conditionals using λ-calculus.

But first... how do we write functions with more than one argument?

$\lambda x\ y.\ x$ ← what we want

$(\lambda x\ y.\ x)$ apple banana

$((\underbrace{\lambda x.\ ((\lambda y.\ x))}_{\text{function being applied to apple}}\ \text{apple})$ banana

$(\lambda y.\ \text{apple})$ banana

apple

Instead of a function that takes multiple arguments, we have:
a function that takes the 1st argument and returns
a function that takes the 2nd argument
(...and returns
a function that takes the 3rd argument, etc.)

let TRUE = $\lambda x.\ (\lambda y.\ x)$
let FALSE = $\lambda x.\ (\lambda y.\ y)$
(A classic way to encode Booleans in lambda calculus.)
Known as a Church encoding.

if <conditional> then <branch 1> else <branch2>

let ITE = $\lambda b.\ (\lambda x.\ (\lambda y.\ \boxed{(b\ x)\ y}))$

"if then else"

How to read this:
$(b\ x)\ y$
Apply b to x, then apply the result of that to y.