CSE 114A lecture 2

agenda:
- ✓ syntax, semantics, pragmatics
- ✓ intro to lambda calculus
- 5-10 minute break + quiz
- intro to Elsa

---

an analogy

human languages: Linguistics ::
programming languages : the field of PL

↑ e.g. Python, Java, Rust, C, Haskell, etc.

syntax — what languages and programs look like

* semantics — what languages and programs mean

pragmatics — how languages and programs are used

---

lambda calculus

↑ (lambda) ↑ "a system of reasoning"

λ ← Greek letter lambda

1936 — Turing machines were invented (Alan Turing)

1936 — lambda calculus was invented (Alonzo Church)

Both universal models of computation!

In lambda calculus, everything is a function.

Traditional (paper) syntax	Elsa syntax	Python
$\lambda x. x$	`\x -> x`	`lambda x: x`
		`def identity(x): return x`

What does this mean?

λ x . x

- binder (λ)
- formal parameter being bound by the binder (x)
- just a delimiter (.)
- body of the function (x)

$\lambda y. y$

```
def identity(y):
    return y
```

$\lambda y. y$ is the same as $\lambda x. x$

$\lambda z. (\lambda x. x)$ — A function that takes an argument, ignores it, and returns the identity function. (body: $(\lambda x. x)$)

$\lambda y. (\lambda x. x)$ — Same.

$(\lambda x. x)\ 3$ — function, argument

```
> identity(3)
3
```

The syntax for calling functions in lambda calculus is incredibly lightweight.

$(\lambda x. x)\ 3$ — function application

$\lambda f. f (\lambda x. x)$ — What does this do? (argument: f; body: $f\ (\lambda x. x)$)

$f\ \ \lambda x. x$

$(\lambda f. f\ (\lambda x. x))\ \ (\lambda x. x)$ — function, argument

$(\lambda x. x)\ \ (\lambda y. y)$ — function, argument

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

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

$\lambda y. y$ and we're done!

The essence of computation in lambda calculus is substitution.

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

```
> identity(3)
3
```
function call

The substitution rule aka the β rule ← Greek letter beta

$(\lambda x. e_1)\ e_2 \rightarrow_\beta e_1[x := e_2]$ ("steps to")

function, argument

"e_1, but with all occurrences of x replaced with e_2"

In lambda calculus, function application is left-associative

$f\ g\ x$ means "apply f to g, get back a value (which is a function), and then apply that function to x"

in other words: $f\ g\ x$ is shorthand for $(f\ g)\ x$

(it's not $f\ (g\ x)$!)

Let's try using the β-rule

$(\lambda x. e_1)\ e_2 \rightarrow_\beta e_1[x := e_2]$

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

$(\lambda x. x)\ (\lambda y. y) \rightarrow_\beta x[x := \lambda y. y]$

($e_1 = x$, $e_2 = \lambda y. y$)

$= \lambda y. y$

What if we wanted functions that take more than one argument?

$\lambda x\ y. x$ (A function that takes two arguments and returns the first one)

$\lambda x. (\lambda y. x)$ (A function that takes an argument and returns a function that returns the original argument)

$(\lambda x. (\lambda y. x))\ 5$

Remember the β-rule:

$(\lambda x. e_1) e_2 \rightarrow_\beta e_1[x := e_2]$

$(\lambda x. (\lambda y. x))\ 5$

$\rightarrow_\beta (\lambda y. x)[x := 5]$

$= \lambda y. 5$ ← the constant function that returns 5!

So $\lambda x. (\lambda y. x)$ is a machine for creating constant functions. cool!

$\lambda x. (\lambda x. x)$

- this occurrence of x is bound by
- this formal parameter
- not this one.

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

$(f\ g)\ x$

$f\ (g\ x)$