Section 4

Plan:

① Review λ-calculus
② Encoding natural numbers in λ
③ Encoding recursion in λ

* * *

λ calculus is a Turing complete programming language (PL)
- can simulate a Turing machine.
- can simulate any modern PL, like C, Java, Python, OCaml, etc.

The syntax of λ calculus consists of
1. Variables: $x, y, z, \ldots$
2. Function abstraction: $\lambda x.x$
3. Function application: $(\lambda x.x)\ y$

The semantics of λ calculus can be given by:

how a λ calculus program/term should be executed/evaluated.

```
let rec eval (t:term) :term =
  match t with
  | v → v
  | λv.t' → λv.t'
  | t1 t2 →
    ( match eval t1, eval t2 with
      | λx.t1', t2' → replace v with t2' in t1'
      | _ , _ → stuck )
```

subst v t_2' t_1'

$[v \mapsto t_2']\ t_1'$

The semantics of λ-calculus terms crucially depend on the substitution function

$$[v \to t_v]\ t$$

Attempt #1

$[v \to t_v]\ t$ = match t with

| u → if $u \neq v$ then t_v else t

| $(t_1\ t_2)$ → $([v \to t_v]\ t_1,\ [v \to t_v]\ t_2)$

| $\lambda u.\ t'$ → $\lambda u.\ [v \to t_v]\ t'$

Problem ?

What if $v = u$, e.g.

$\text{eval}\ (\lambda x.\ \lambda x.\ x)\ y = [x \mapsto y]\ (\lambda x.\ x)$

$\Downarrow$

$\lambda x.\ x \qquad = \lambda x.\ y$

Functions are equiv up to renaming.

Attempt #2

$[v \to t_v]\ t$ = match t with

| u → ⋯

| $(t_1\ t_2)$ → ⋯

| $\lambda u.\ t'$ → if $v \neq u$ then $\lambda u.\ [v \to t_v]\ t'$

else t.

Problem ?

What if u is free in t_v, e.g.

$\text{eval}\ (\lambda y.\ \lambda x.\ y)\ x = [y \mapsto x]\ (\lambda x.\ y)$

$\|$

$(\lambda y.\ \lambda z.\ y)\ x \quad = \lambda x.\ x$

$\Downarrow$

$\lambda z.\ x$

A variable is free if it isn't a formal param.

Attempt #3 : $[v \to t_v]\, t$ = match t with

...

| $\lambda u.\, t' \to$

if $u \neq v$ then

if $u \notin FV(t_v)$ then

$\lambda u.\, [v \to t_v]\, t'$

else

?

else t .

Problem?

We're stuck in the second case !

eval $(\lambda y.\, \lambda x.\, y)\, x = [y \mapsto x]\, \lambda x.\, y$

We need to perform α-renaming .

Encoding things in λ-calculus.

- Booleans : true = $\lambda x.\lambda y.x$

  false = $\lambda x.\lambda y.y$

  if-then-else = $\lambda b.\lambda x.\lambda y.\ b\ x\ y$

- Integers :

n ⟷ (encode / decode) apply a function n times

(demo)

		apply f
0	$\lambda f.\lambda x.x$	0 times
1	$\lambda f.\lambda x.f\ x$	1 time
2	$\lambda f.\lambda x.f\ (f\ x)$	2 times
3	$\lambda f.\lambda x.f\ (f\ (f\ x))$	
⋮	⋮	
n	$\lambda f.\lambda x.\underbrace{f\ (\cdots\ (f}_{n \text{ times}}\ x))$	

Church encoding

Addition : (church encoding of $a+b$)

$\lambda a.\lambda b.(\lambda f.\lambda x.\ b\ f\ (a\ f\ x))$

"$(a\ f)\ x$" applies f to x a times

"$(b\ f)\ y$" applies f to y b times

"$b\ f\ (a\ f\ x)$" applies f to x $(a+b)$ times

Note: The encode & decode functions are not part of λ-calculus; they just help us read an encoding more easily

Multiplication:

$$\lambda a.\, \lambda b.\, (\lambda f.\, \lambda x.\, \underbrace{b\ (a\ f)\ x}_{\text{applies } f \text{ to } x \text{ } a \cdot b \text{ times}})$$

Hint: Partial application.

Exponentiation:

$$\lambda a.\, \lambda b.\, (\lambda f.\, \lambda x.\, \underbrace{(b\ a)\ f\ x}_{\text{applies } f \text{ to } x \text{ } a^b \text{ times}})$$

Encoding let expressions:

let $x = e_1$
in e_2

"Substitute x for e_1 in e_2"

$$[x \mapsto e_1]\, e_2$$

$$\Uparrow$$

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

Encoding recursion

Consider our favorite recursive function, factorial:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 1$$

```
let fact = λn.
    if n=0 then 1
    else n * fact(n-1)
```

Cannot define this in λ because the recursive call to fact is an unbound reference.

* But we can define what's called the generator for a recursive function:

```
mystery = λf . λn.
    if n=0 then 1 else f * (n-1)
```

"Lifting the recursive call into a formal parameter"

Then, fact = mystery (mystery (... (mystery bad))) (∞ many times)

e.g. λx.0

But programs are finite.

Hope: There's a magical term "magic" in λ-calculus s.t. (magic mystery) behaves just like fact.

What properties does "magic" have?

Properties of "magic"

magic mystery ⟺ fact < hopefully >

= λn. if n=0 then 1 < expand def of fact >
  else n * fact (n-1)

= λn. if n=0 then 1 < by our hope >
  else n * (magic mystery) (n-1)

mystery = λf. λn.
  if n=0 then 1
  else n * f (n-1)

= (λf. λn. if n=0 then 1
  else n * f (n-1)) (magic mystery)
  < by λ-abstraction and application >

= mystery (magic mystery) < by def of mystery >

magic mystery ⟺ mystery (magic mystery)

1. (⟹) Tells us the behavior of (magic mystery)

   It unrolls the body of mystery by supplying a copy of itself as the argument.

   magic mystery = (λn. if n=0 then 1
   else n * (magic mystery) (n-1))

   Recursion via self-replication.

magic mystery = mystery (magic mystery)

2. ($\Leftarrow$) Let's say x = magic mystery.

Then mystery(x) = x .

x is called a fixed-point of function mystery.

If f is a function, and $f(x) = x$ for some x,
then x is called a fixed-point of f

Thus, magic "computes" the fixed-point of any generator.
In λ-calculs, `magic` are the fixed-point combinators,
such as the Y-combinator and Z-combinator:
$(\lambda f.\ \lambda x.\ f(x\ x))\ (\lambda f.\ \lambda x\ \lambda f.((\lambda x.\ f(\lambda y.\ x\ x\ y))(\lambda x.\ f(\lambda y.\ x\ x\ y))$

Summary, to encode a recursive function f in λ-calculus

1. Write down its generator, which is non-recursive. Call it g.
2. Apply your favorite fixed-point combinator $\mathcal{F}$ to the generator g.
3. The application $\mathcal{F}g$ is a term in λ-calculus that behaves just like the recursive function f.

Example.
f = fact
g = mystery
$\mathcal{F}$ = magic
$\mathcal{F}g$ = magic mystery
$\Leftrightarrow$ fact

Recursive functions are fixed points of their generators

Note: in λ^+, magic is the built-in fix operator.