CSE114A - lecture 3

agenda:

- recap: β-rule ✓
- λ-calc syntax ✓
- Redexes and normal form
- How to encode things we want in lambda calculus
  - Booleans (true, false)
  - Boolean operations (and, or, not, ...)
  - ~~conditionals~~ (if ... then ... else)
  - more if time!

The β-rule

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

"e1, but with occurrences of x replaced with the argument e2" *

binder, body

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

function: `\x -> x`; argument: `\y -> y`; `= \y -> y`

Let's write a function (if we can) that takes two arguments, and returns the first one.

`\x -> (\y -> x)` ← A machine for making constant functions!

`(\x -> (\y -> x)) "yuki"`

Function: `\x -> (\y -> x)`; argument: `"yuki"`

Quiz question 1: Apply this function to this argument using the β-rule. What do you get?

`=b> \y -> "yuki"` ← The constant function that returns "yuki"!

one β-step

`((\x -> (\y -> x)) "yuki") "islay"`

`=*> "yuki"`

any number of β-steps

λ-calculus syntax

3 Syntactic Forms:

- variables,
- function definitions (aka lambda abstraction)
- function calls (aka application)

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

↑ expression

Notational conventions

- The body of a lambda abstraction extends as far right → as possible.

Lecture quiz question 2:
does `\x -> m n` mean:

(a) `\x -> (m n)`

or

(b) `(\x -> m) n`

?

it's (a)!

- Applications are left-associative:
  `p q r` means `(p q) r`,
  not `p (q r)`.

Think of "p q r" as being a call to the function p, but with two arguments... but p gets them one at a time.

(This is called currying!)

- Instead of, e.g.,
  `\x -> (\y -> (\z -> e))`,
  as a shorthand we can write
  `\x -> \y -> \z -> e`
  and then further shorten it to
  `\x y z -> e`

Putting together all these conventions:

`(((\x -> (\y -> (\z -> z))) q) r) s`

can be concisely written as:

`(\x y z -> z) q r s`

Let's evaluate this:

a place to use the β-rule!

`(\x y z -> z) q r s`

`=b> (\y z -> z) r s`

`=b> (\z -> z) s`

`=b> s`

"A place to use the β-rule" is a reducible expression, also known as a redex.

A redex is an expression that can be reduced by taking a `=b>` step!

A λ-calc expression in normal form is one that has no redexes.

You might have options for where to take a `=b>` step!

a redex!

`(\x y -> (\z -> z) x) "yuki" "islay"`

a redex!

`=b> (\x y -> x) "yuki" "islay"`

or, we could've done...

`=b> (\y -> (\z -> z) "yuki") "islay"`

It doesn't matter which order of evaluation you pick — the eventual result will be the same!