```
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
```

"make"

$$(\lambda x \to x) \quad z$$

QUIZ

Free

In the expression `(\x -> x) x`, is `x` *bound* or *free*?

1st 2nd

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)        = ???` $\{x\}$

`FV(\x -> e)  = ???` $FV(e) - \{x\}$

`FV(e1 e2)    = ???` $FV(e_1) \cup FV(e_2)$

$$\underbrace{((\lambda x \to x)\ z)}_{e_1}\ \underbrace{dog}_{e_2}$$

\x → x

"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*)

$e \to e_1 \to e_2 \to e_3 \to e_{final}$

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 -> BODY) ARG

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

BODY [f := ARG]

QUIZ

\x -> BODY ARG

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

A. `apple`

B. `\y -> apple`

C. `\x -> apple`

D. `\y -> y`

E. `\x -> y`

QUIZ

BODY ARG

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

apple (\x→x)

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

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

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

D. `apple`

E. `\x -> x`

A Tricky One

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

free vars DIFF THAN params

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 then???
```

Rewrite Rules of Lambda Calculus

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

$$(\lambda x \to e)$$

$$\Rightarrow_a (\lambda y \to e[x := y])$$

$\lambda a \to a$ $\quad$ $\lambda zebra \to zebra$

$\lambda b \to b$

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 terms are **not** in *normal form* ?

can still be reduced

A. `x`

no Lambda!

B. `x y`

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

D. `x (\y -> y)` NOT a redex (because λ is on RIGHT)

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
```

BODY arg

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

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

\ab → a(cab)

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!

(\a -> (\b -> a)) (c a b)

(\b -> (c a b))

oh no!

"b" captured!!!

$(\backslash x \to e_1) \; e_2 \; =b> \; e_1[x := e_2]$

"formal" "body" "arg"

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.

if COND
    STUFF
ELSE
    OTHER

let IF = \cond stuff other ->

(NOT TRUE) = FALSE
(NOT FALSE) = TRUE

NOT = \b -> IF b FALSE True

NOT = \b -> IF False

AND = \b1 b2 -> IF b1 b2 False

OR = \b1 b2 -> IF b1 TRUE b2

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?

1 "cat"

[10, 20, 30]

Programming in λ-calculus

- **Booleans** [done]
- $\boxed{\text{Records (structs, tuples)}}$
- Numbers
- **Functions** [we got those]
- Recursion

$$O = \{ fst : \underline{1}, snd : \underline{\text{"cat"}}, thd : [10, 20, 30] \}$$

$O.fst$
$O.snd$
$O.thd$

$$FST\ ((pack\ V_1)\ V_2) = V_1$$
$$SND\ ((pack\ V_1)\ V_2) = V_2$$

HW → SUNDAY 4/14 23:59

`(\x y z -> e) a1 a2 a3`

`(((\x -> (\y -> (\z -> e))) a1) a2) a3`

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.