```
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(e1) ∪ FV(e2)
```

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

$$\backslash x \to x$$

"*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*)

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

$$\overset{a}{\Rightarrow} (\lambda y \to e[x := y])$$

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

$\lambda b \to b$

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. $\boxed{\texttt{x}}$

no Lambda!

B. $\boxed{\texttt{x y}}$

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

D. `x (\y -> y)` NOT a redex (because 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)
  =?> ???
```

Elsa shortcuts

Named λ-terms: