

Syntactic Sugar

$$e ::= x \mid (\lambda x. e) \mid (e_1 e_2)$$

\uparrow \uparrow \uparrow
 var func call

instead of	we write
$\lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow e))$	$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$
$\lambda x \rightarrow \lambda y \rightarrow \lambda z \rightarrow e$	$\lambda x y z \rightarrow e$
$((e_1 e_2) e_3) e_4$	$e_1 e_2 e_3 e_4$

$(\lambda x \rightarrow (\lambda y \rightarrow y))$

$\lambda x y \rightarrow y$

-- A function that takes two arguments
-- and returns the second one...

$(\lambda x y \rightarrow y)$ ^x apple ^y banana -- ... applied to two arguments

$((\lambda x \rightarrow (\lambda y \rightarrow y)) \text{ apple}) \text{ banana}$

\hookrightarrow^* banana

Semantics: What Programs Mean

How do I “run” / “execute” a λ -term?

Think of middle-school algebra:

-- Simplify expression:

$$\begin{aligned}
 & (1 + 2) * ((3 * 8) - 2) \\
 = & \\
 & 3 * ((3 * 8) - 2) \\
 = & \\
 & 3 * (24 - 2) \\
 = & \\
 & 3 * 22 \\
 = & \\
 & 66
 \end{aligned}$$

$$\begin{aligned}
 & (1+2) * ((3*8) - 2) \\
 = & 3 * ((3*8) - 2) \\
 = & 3 * (24 - 2) \\
 = & 3 * 22 \\
 = & \boxed{66}
 \end{aligned}$$

Execute = rewrite step-by-step

- Following simple rules
- until no more rules apply

Rewrite Rules of Lambda Calculus

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

But first we have to talk about **scope**

```
{ int x;
  ...
}
```

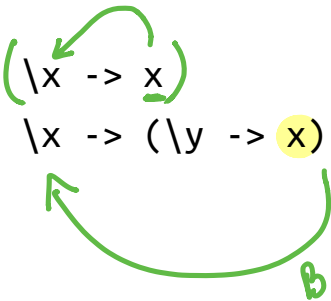
Semantics: Scope of a Variable

The part of a program where a **variable is visible**

In the expression $(\lambda x \rightarrow e)$

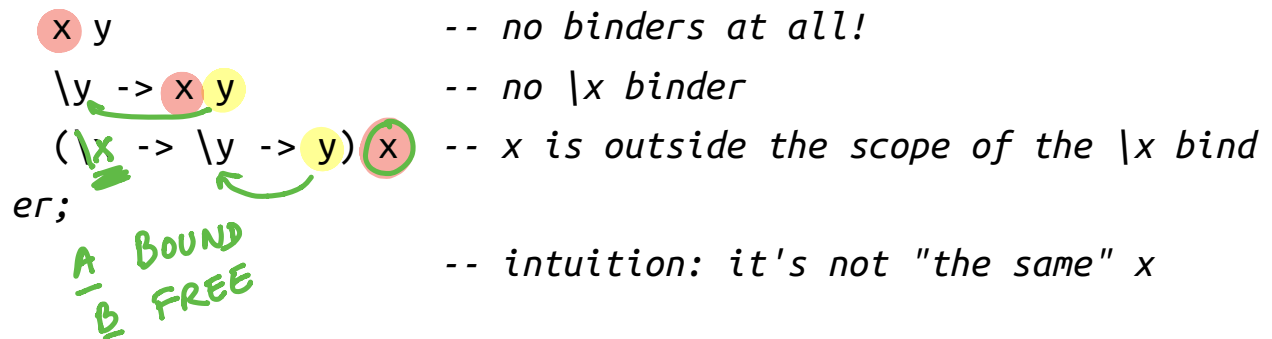
- x is the newly introduced variable
- e is the scope of x
- any occurrence of x in $\lambda x \rightarrow e$ is bound (by the binder λx)

For example, x is bound in:



An occurrence of x in e is free if it's not bound by an enclosing abstraction

For example, x is free in:



QUIZ

In the expression $(\lambda x \rightarrow x) x$, is x bound or free?

- A. first occurrence is bound, second is bound
- ✓ B. first occurrence is bound, second is free
- C. first occurrence is free, second is bound
- D. first occurrence is free, second is free

FREE OCC of x

EXERCISE: Free Variables

$$FV(\lambda x \rightarrow x) = FV(x) = \{x\}$$

$$FV(\lambda x \rightarrow (\lambda y \rightarrow z)) = FV(\lambda y \rightarrow z) = FV(z) = \{z\}$$

x, y, z

An variable x is **free** in e if there exists a free occurrence of x in e

y is free ($\lambda x \rightarrow x$) NO

We can formally define the set of all free variables in a term like so:

y is free ($\lambda y \rightarrow y$) NO

$$\left[\begin{array}{l} \text{FV}(x) = ??? \{x\} \\ \text{FV}(\lambda x \rightarrow e) = ??? \text{FV}(e) - x \\ \text{FV}(e_1 e_2) = ??? \text{FV}(e_1) \cup \text{FV}(e_2) \end{array} \right.$$

y is free ($\lambda x \rightarrow y$) YES

$\text{FV} : \text{Expr} \rightarrow \underline{\text{Set of free vars in Expr}}$
 not expr

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

Semantics: Redex

A **redex** is a term of the form

$(\lambda x \rightarrow e1) e2$

A function $(\lambda x \rightarrow e1)$

- x is the *parameter*
- $e1$ is the *returned expression* "body"

Applied to an argument $e2$

- $e2$ is the *argument*

Semantics: β -Reduction

A redex β -steps to another term ...

$$(\lambda x \rightarrow e_1) e_2 \rightarrow e_1[x := e_2]$$

↑ free occ of x replaced by e₂

where $e_1[x := e_2]$ means

“ e_1 with all *free* occurrences of x replaced with e_2 ”

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 $(\lambda x \rightarrow e_1) e_2$ β -steps to $e_1[x := e_2]$

Redex Examples

$(\underline{\lambda}x \rightarrow \underline{x})$ apple

$(\lambda x \rightarrow x)$ apple
 $=b>$ apple

Is this right? Ask Elsa (<http://goto.ucsd.edu:8095/index.html#?demo=blank.lc>)!

QUIZ

$(\lambda x \rightarrow (\lambda y \rightarrow y))$ apple
 $=b>$???

formal *body* *arg*

$(\lambda x \rightarrow (\lambda y \rightarrow (y \text{ apple})))$

A. apple

B. $\lambda y \rightarrow$ apple

body [formal := arg]

$(\lambda y \rightarrow y)$

C. `\x -> apple`

D. `\y -> y`

E. `\x -> y`

QUIZ

$(\lambda x \rightarrow y\ x\ y\ x)\ \text{apple}$
 =b> ???

$\text{body [form := args]}$
 $y\ x\ y\ x\ [x := \text{apple}]$

A. `apple apple apple apple`

B. `y apple y apple` ✓

C. `y y y y`

D. `apple`

QUIZ

$(\lambda x \rightarrow x (\lambda x \rightarrow x)) \text{ apple}$
 =b> ???

Handwritten annotations:
 - λx is labeled "form" (blue arrow)
 - x is labeled "free" (green arrow)
 - $(\lambda x \rightarrow x)$ is labeled "body" (blue arrow)
 - x is labeled "bound" (green arrow)
 - apple is labeled "arg" (blue arrow)

Handwritten diagram:
 $\lambda x (\lambda x \rightarrow x)$
 - λ has a checkmark above it
 - x has a checkmark below it
 - $(\lambda x \rightarrow x)$ has a checkmark below it
 - A blue arrow points from the x in the body to the x in the argument position, labeled "body [form := arg]"

A. $\text{apple } (\lambda x \rightarrow x)$

B. $\text{apple } (\lambda \text{apple} \rightarrow \text{apple})$

C. $\text{apple } (\lambda x \rightarrow \text{apple})$

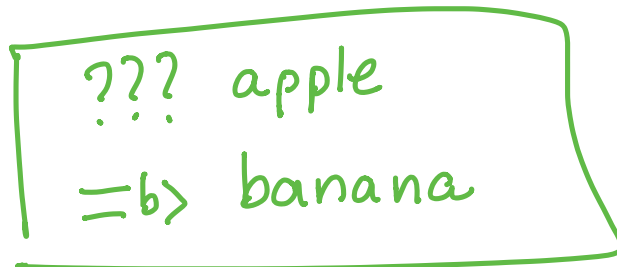
D. apple

E. $\lambda x \rightarrow x$

EXERCISE

What is a λ -term `fill_this_in` such that

`fill_this_in apple`
`=b> banana`



???

apple

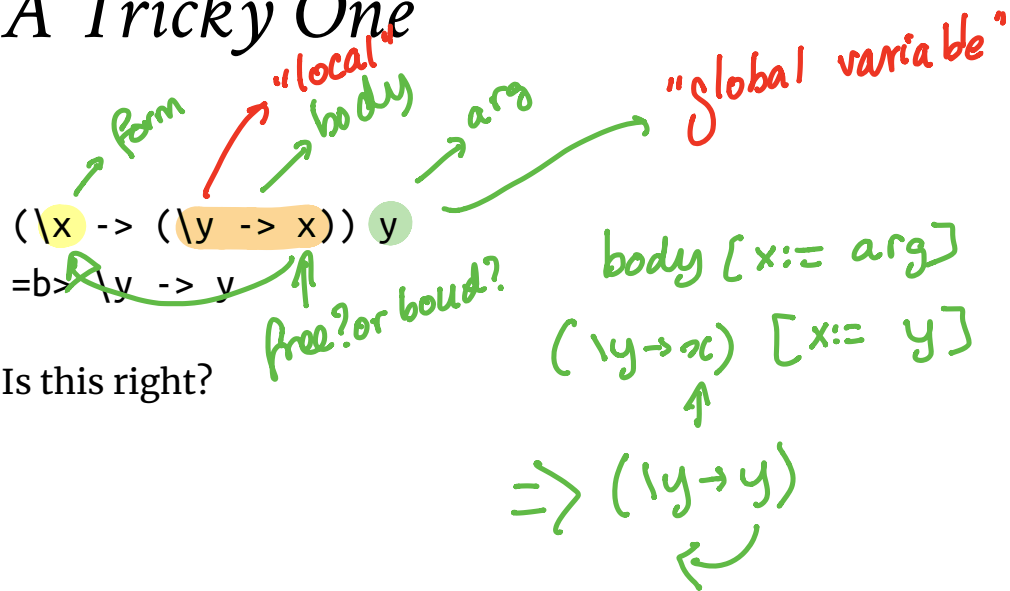
=b>

banana

ELSA: <https://goto.ucsd.edu/elsa/index.html>

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434473_24432.lc)

A Tricky One



Something is Fishy

$(\lambda x \rightarrow (\lambda y \rightarrow x)) y$
 =b> $\lambda y \rightarrow y$

Is this right?

Problem: The *free* y in the argument has been *captured* by λy in body! "become bound"

Solution: Ensure that *formals* in the body are *different from free-variables* of argument!

IDE (A) YES
intellij (B) NEVER
eclipse

Capture-Avoiding Substitution

We have to fix our definition of β -reduction:

$$(\lambda x \rightarrow e1) e2 \quad =_{\beta} \quad e1[x := e2]$$

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

Formally:

$$x[x := e] = e$$

$$y[x := e] = y \quad \text{-- as } x \neq y$$

$$(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$$

$$(\lambda x \rightarrow e_1)[x := e] = \lambda x \rightarrow e_1 \quad \text{-- Q: Why?}$$

leave `e1` unchanged?

$$(\lambda y \rightarrow e_1)[x := e] \\ | \text{ not } (y \text{ in } FV(e)) = \lambda y \rightarrow e_1[x := e]$$

Oops, but what to do if y is in the free-variables of e ?

- i.e. if $\lambda y \rightarrow \dots$ may *capture* those free variables?

Rewrite Rules of Lambda Calculus

1. β -step (aka *function call*)

2. α -step (aka *renaming formals*)

Semantics: α -Renaming

$\lambda x \rightarrow e \stackrel{a}{=} \lambda y \rightarrow e[x := y]$
 where not (y in FV(e))

$\lambda x \rightarrow y$

- We rename a **formal parameter** x to y
- By **replace all occurrences of x in the body with y**
- We say that $\lambda x \rightarrow e$ α -steps to $\lambda y \rightarrow e[x := y]$

Example:

$\lambda x \rightarrow x =_a \lambda y \rightarrow y =_a \lambda z \rightarrow z$

All these expressions are α -equivalent

What's wrong with these?

-- (A)
 $\lambda f \rightarrow f x =_a \lambda x \rightarrow x x$? *x is free in f x*

-- (B)
 $(\lambda x \rightarrow (\lambda y \rightarrow y) y) =_a (\lambda x \rightarrow \lambda z \rightarrow z) z$
 NOT ALLOW

Tricky Example Revisited

```

(\x -> (\y -> x)) y

```

capture

```

=a> (\x -> (\z -> x)) y

```

ure!

```

=b> \z -> y

```

-- rename 'y' to 'z' to avoid

-- now do b-step without capture!

To avoid getting confused,

- you can **always rename** formals,
- so different **variables** have different **names!**

Normal Forms

Recall **redex** is a λ -term of the form

$(\lambda x \rightarrow e_1) e_2$

} REDEX

A λ -term is in **normal form** if it contains no redexes.

$(2+3) * 5$
 \uparrow
 redex

22
 normal-form
 no redex!

QUIZ

Which of the following term are **not** in normal form?

A. x *no red*

B. $x y$ *no red*

C. $(\lambda x \rightarrow x) y$ *yes red*

D. $x (\lambda y \rightarrow y)$ *no red*

E. C and D

ie contain a redex

$(\lambda x \rightarrow e_1) e_2$
 \uparrow left \uparrow right

$$e \xRightarrow{a} e_1 \xRightarrow{b} e_2 \xRightarrow{a} e_3 \xRightarrow{a} \dots \Rightarrow \underbrace{e'}_{\text{normal form}}$$

Semantics: Evaluation

A λ -term e evaluates to e' if

1. There is a sequence of steps

$$e \Rightarrow e_1 \Rightarrow \dots \Rightarrow e_N \Rightarrow e'$$

where each \Rightarrow is either \xRightarrow{a} or \xRightarrow{b} and $N \geq 0$

2. e' is in *normal form*

Examples of Evaluation

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

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

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

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) 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 $=d>$
- $e1 =^{\sim}> e2$: $e1$ evaluates to $e2$ and **$e2$ is in normal form**

EXERCISE

Fill in the definitions of FIRST, SECOND and THIRD such that you get the following behavior in `elsa`

```
let FIRST = fill_this_in
let SECOND = fill_this_in
let THIRD = fill_this_in
```

On Third

```
eval ex1 :
(((FIRST apple) banana) orange)
=> apple
```

```
eval ex2 :
(((SECOND apple) banana) orange)
=> banana
```

```
eval ex3 :
(((THIRD apple) banana) orange)
=> orange
```

ELSA: <https://goto.ucsd.edu/elsa/index.html>

Click here to try this exercise (https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434130_24421.lc)

Non-Terminating Evaluation

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

Some programs loop back to themselves...

... and *never* reduce to a normal form!

This combinator is called Ω