

## **Semantics: Redex**

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

( $\lambda x \rightarrow e_1$ )  $e_2$

# REDEX

A function  $(\lambda x \rightarrow e_1)$

- $x$  is the *parameter*
- $e_1$  is the *returned expression*

Applied to an argument  $e_2$

- $e_2$  is the *argument*

# *Semantics: $\beta$ -Reduction*

A **redex** b-steps to another term ...

$$(\lambda x \rightarrow e_1) \ e_2 \quad =_{\text{b}} \quad e_1[x := e_2]$$

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

## *Redex Examples*

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

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> ??? BODY ARG

Body [x := ARG]



Body

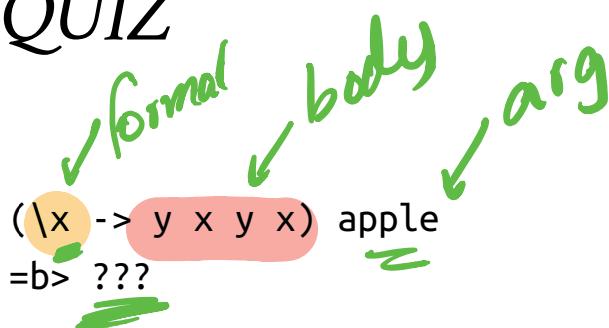
- A. apple
- B.  $\lambda y \rightarrow \text{apple}$
- C.  $\lambda x \rightarrow \text{apple}$
- D.  $\lambda y \rightarrow y$
- E.  $\lambda x \rightarrow y$

# QUIZ

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

formal body arg

=b> ???



NEW

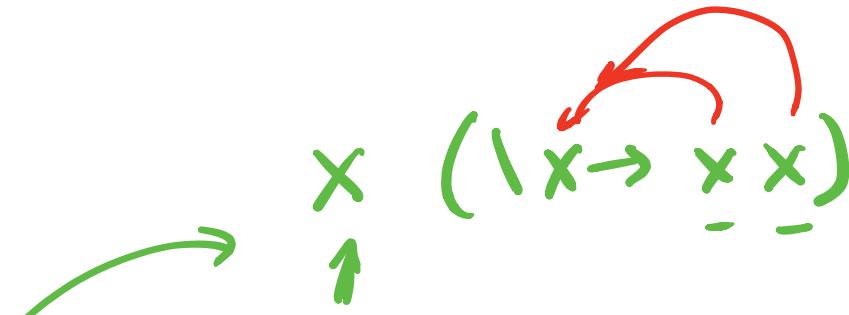
- A. apple apple apple apple

- B. y apple y apple



- C. y y y y
- D. apple

# QUIZ



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

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.  $\text{apple}$

E.  $\lambda x \rightarrow x$

# *EXERCISE*

What is a  $\lambda$ -term `fill_this_in` such that

```
fill_this_in 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](https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434473_24432.lc))

## A Tricky One

$$\begin{aligned} & (\lambda x \rightarrow (\lambda y \rightarrow x)) y \\ & = b> \lambda y \rightarrow y \end{aligned}$$

*DIFF*      *FREE*

*λy → y*      *= BOUND*

Is this right?

*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 *\y in body!*

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

# *Capture-Avoiding Substitution*

We have to fix our definition of  $\beta$ -reduction:

$$(\lambda x \rightarrow e_1) e_2 =_{\beta} e_1[x := e_2]$$

where  $e_1[x := e_2]$  means “ ~~$e_1$  with all free occurrences of  $x$  replaced with  $e_2$~~ ”

- $e_1$  with all *free* occurrences of  $x$  replaced with  $e_2$
- as long as no free variables of  $e_2$  get captured

Formally:

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

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

$$(e1\ e2)[x := e] = (e1[x := e]) (e2[x := e])$$

$$(\lambda x \rightarrow e1)[x := e] = \lambda x \rightarrow e1 \quad \text{-- Q: Why leave `e1' unchanged?}$$

$$(\lambda y \rightarrow e1)[x := e] \\ | \text{ not } (y \text{ in } FV(e)) = \lambda y \rightarrow e1[x := e]$$

DEF

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.  $\beta$ -step (aka *function call*)
2.  $\alpha$ -step (aka *renaming formals*)

$$\begin{array}{ccc}
 (\lambda x \rightarrow (\lambda y \rightarrow x)) & & y \\
 \downarrow \text{"a-step"} & & \\
 (\lambda x \rightarrow (\lambda n \rightarrow x)) & & y
 \end{array}$$

func (x) { return x+1 }

"a-renam"

func(y) { return y+1 }

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

$\lambda y \rightarrow y$  "capture" y

## Semantics: $\alpha$ -Renaming

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

- We rename a formal parameter  $x$  to  $y$
- By replace all occurrences of  $x$  in the body with  $y$

- We say that  $\lambda x \rightarrow e$   $\alpha$ -steps to  $\lambda y \rightarrow e[x := y]$

Example:

$\text{fun}(y) \{ \text{ret } y \}$

$$\lambda x \rightarrow x =\alpha> \lambda y \rightarrow y =\alpha> \lambda z \rightarrow z$$

All these expressions are  $\alpha$ -equivalent

$\text{fun}(x) \{ \text{ret } x \}$

$\text{fun}(z) \{ \text{ret } z \}$

What's wrong with these?

-- (A)

$$\lambda f \rightarrow f x =\alpha> \lambda x \rightarrow x x$$

*Do not replace  
the FREE 'y'*

-- (B)

$$(\lambda x \rightarrow \lambda y \rightarrow y) y =\alpha> (\lambda x \rightarrow \lambda z \rightarrow z) z$$

# *Tricky Example Revisited*

$$\begin{aligned} & (\lambda x \rightarrow (\lambda y \rightarrow x)) y \\ \Rightarrow & \underline{(\lambda x \rightarrow (\lambda z \rightarrow x)) y} \\ \Rightarrow & \underline{\lambda z \rightarrow y} \end{aligned}$$

-- rename 'y' to 'z' to avoid capture  
-- now do b-step without capture!

To avoid getting confused,

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

*Normal Forms* = Has No Redex

Recall **redex** is a  $\lambda$ -term of the form

$(\lambda x \rightarrow e_1) e_2$

A  $\lambda$ -term is in **normal form** if it *contains no redexes*.

# QUIZ

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

A.  $x \quad \text{NF}$

ie do NOT CONTAIN ANY  
REDEX

B.  $x \ y \quad \text{NF}$

C.  $(\lambda x \ -> x) \ y \quad \text{not NF}$

                          
 $(\lambda x \rightarrow e_1) \ e_2$

D.  $x \ (\lambda y \ -> y) \quad \text{NF}$

~~E. C and D~~

## *Semantics: Evaluation*

A  $\lambda$ -term  $e$  evaluates to  $e'$  if

1. There is a sequence of steps

$$\begin{array}{l} e \\ \xrightarrow{?} e_1 \\ \xrightarrow{?} e_2 \\ \vdots \end{array}$$

$e =?> e_1 =?> \dots =?> e_N =?> e'$

$\Rightarrow e'$   
↑  
NF

where each  $=?>$  is either  $=a>$  or  $=b>$  and  $N \geq 0$

2.  $e'$  is in *normal form*

## Examples of Evaluation

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

$$\begin{aligned} & (\lambda f \rightarrow f (\lambda x \rightarrow x)) (\lambda x \rightarrow x) \\ &=?> ??? \end{aligned}$$
$$\begin{aligned} & (\lambda x \rightarrow x x) (\lambda x \rightarrow x) \\ &=?> ??? \end{aligned}$$

## *Elsa shortcuts*

Named  $\lambda$ -terms:

**let** ID =  $\lambda x \rightarrow x$  -- *abbreviation for*  $\lambda x \rightarrow x$

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 =~> 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 else

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

```
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](https://goto.ucsd.edu/elsa/index.html#?demo=permalink%2F1585434130_24421.lc))

# *Non-Terminating Evaluation*

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

Some programs loop back to themselves...

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

This combinator is called  $\Omega$

What if we pass  $\Omega$  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!

# *Programming in $\lambda$ -calculus*

*Real languages have lots of features*

- Booleans ✓
- Records (structs, tuples) ✓
- Numbers ✓
- Functions [we got those]
- Recursion ✓

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

Lets see how to *encode* all of these features with the  $\lambda$ -calculus.