



LAMBDA CALCULUS SYNTAX AND SEMANTICS

CSE 340 FALL 2021



Rida Bazzi



Lambda Calculus

We can think of lambda calculus as a basic programming language that has

- Syntax that tells us how to
 - parse Lambda expressions, and
 - identify reducible expressions
- Semantics that tell us how to “execute” lambda expressions. This involves the following
 - determine bindings (scoping rules)
 - do α -renaming (if needed)
 - apply β -reductions
 - deciding on order of evaluation

In the following notes, I cover each of the points above in detail.

Syntax

The grammar for lambda calculus is the following

$t \rightarrow$	x	// x is the name of a variable from // a set of variable names
$t \rightarrow$	$\lambda x. t$	// where x is the name of a variable // this is called an abstraction // t in $\lambda x. t$ is the body of the abstraction
$t \rightarrow$	$t t$	// this is called an application
$t \rightarrow$	(t)	// parentheses

This grammar is clearly ambiguous. There are two source of ambiguity

1. Identifying the body of abstractions for the rule $t \rightarrow \lambda x. t$

Consider the input $\lambda x. x x$. There are two way to parse this input. Here are two leftmost derivations

$$\begin{aligned} t &\Rightarrow t t \Rightarrow \lambda x. t t \Rightarrow \lambda x. x t \Rightarrow \lambda x. x x & (1) \\ t &\Rightarrow \lambda x. t \Rightarrow \lambda x. t t \Rightarrow \lambda x. x t \Rightarrow \lambda x. x x & (2) \end{aligned}$$

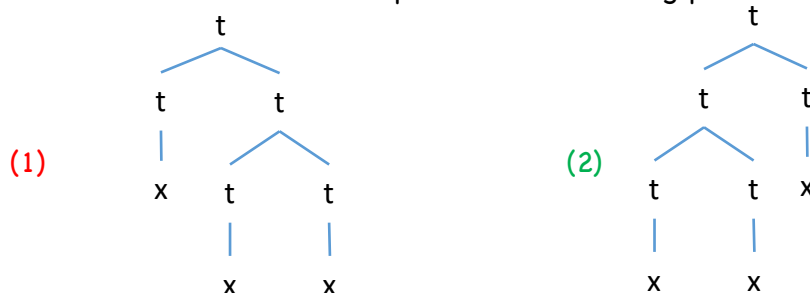
In derivation (1) $\lambda x. x$ is a term by itself and the second x is another term. In derivation (2), $x x$ is one term which forms the body of the abstraction. By the disambiguation rule, that we will consider next, derivation (2) is the correct way to parser it

2. Grouping terms for the $t \rightarrow t t$ rule.

Consider the input $x x x$. There are two ways to parse it. Here are two left most derivations:

$$\begin{aligned} t &\Rightarrow t t \Rightarrow x t \Rightarrow x t t \Rightarrow x x t \Rightarrow x x x & (1) \\ t &\Rightarrow t t \Rightarrow t t t \Rightarrow x t t \Rightarrow x x t \Rightarrow x x x & (2) \end{aligned}$$

These two derivations correspond to the following parse trees



By the disambiguation rule, that we will consider next, the right way to group is to group according to the tree number (2).

Syntax: Disambiguation Rules

Typically, when we have an ambiguous grammar, we disambiguate it in one of two ways:

- propose another grammar that is not ambiguous
- give disambiguation rules to uniquely parse of λ -expressions, also called terms (I will use terms and expressions interchangeably)

We will use disambiguation rules. The rules for disambiguation are the following:

1. **Abstractions extend as far to the right as possible without crossing enclosing parentheses.** Given a $\lambda x.$, the body of the abstraction on $\lambda x.$ extends as far to the right as possible without crossing a right parenthesis which is part of a pair of matching parentheses that enclose the $\lambda x.$
2. **Application is left-associative.**

We examine each rule in details on the following pages

Determining the body of an abstraction

1. An Abstraction on $\lambda x.$ extends as far to the right as possible without crossing a right parenthesis which is part of a pair of matching parentheses that enclose the $\lambda x.$

The goal here is to determine the body of the abstraction, where the body of an abstraction of the form $\lambda x.t$ is the term t .

Given a λ -expression which can have multiple abstractions, we determine the bodies of abstractions as follows:

- If $\lambda x.$ is not between any matching pair of parentheses, the body of the abstraction extends from the $\lambda x.$ to the end of the expression.
- If $\lambda x.$ is between a matching pair of parentheses, then we consider the innermost matching pair of parentheses around the $\lambda x.$ The body of the abstraction extends from the $\lambda x.$ to the right parenthesis of this innermost matching pair

Examples: the examples show the bodies of selected abstractions

1. $x\ x\ x\ \lambda x. x\ (\ (\lambda x. x\ (\lambda x. x\ \lambda x. x\)\ x)\ x)\ x$

No parentheses enclose $\lambda x.$
Body extends all the way to the right

2. $\lambda x. x\ (\lambda x. x\ (\lambda x. x\ (\lambda x. x\ \lambda x. x\)\ x)\ x)\ x$

innermost matching pair of
parenthesis surrounding $\lambda x.$

body for $\lambda x.$

3. $x\ (x\ x\ \lambda x. x\ (\lambda x. x\ (\lambda x. x\ \lambda x. x\)\ x)\ x)\ x$

innermost matching pair of
parenthesis surrounding $\lambda x.$

Determining the body of an abstraction

More examples:

innermost matching pair of parentheses surrounding λx .

4. $x (\lambda x. x x (x \lambda x. x x (\lambda x. x \lambda x. x) x) x) x) x$

body for λx .

innermost matching pair of parentheses surrounding λx .

5. $x (\lambda x. x (x \lambda x. x \lambda x. x))$

body for λx .

6. $\lambda x. x (x \lambda x. x x) x$

body for λx .

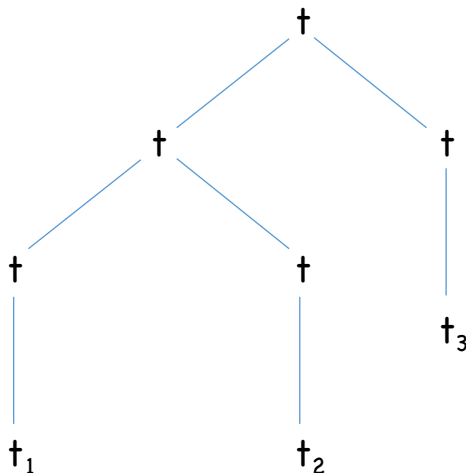
Grouping applications

2. Application is left-associative

We need to determine how to group applications.

This step is done after we have determined the bodies of all abstractions.

The rule to follow is: application is left associative. If t_1 , t_2 and t_3 are three terms in the expression $t_1 t_2 t_3$, then $t_1 t_2 t_3$ is parsed as



You can think of the goal of this step as adding parentheses to achieve the correct grouping

This is explained further on the next page

Note. The expression should be of the form $t_1 t_2 t_3$ **after identifying the bodies of the abstractions**. For example, consider the expression

$$\lambda x. x \ \lambda x. x \ \lambda x. x .$$

This expression is not of the form $t_1 t_2 t_3$ where t_1 , t_2 and t_3 are terms because after we identify the body of abstractions, we find that the expression is of the form $\lambda x. t$, where

$$t = x \ \lambda x. x \ \lambda x. x$$

Parsing General Expressions

Here are the steps for grouping terms in a general lambda expression

1. Identify the bodies of all abstractions
2. If an abstraction does not have parentheses around it, add parentheses around it (note that "around it" means immediately around it).
3. Within the body of each abstraction, group terms using left associativity. In the grouping, treat anything between a pair of parentheses as one term (which it is, according to the grammar).
4. Left associative grouping is also applied for all terms within a pair of matching parentheses.
5. Left associative grouping is also applied at the top level outside all bodies of abstractions

We look at an example

$\lambda x. x \lambda x. x (\lambda x. x) \lambda x. x x \lambda x. x$

1. We identify the bodies of all abstractions and we get

$\lambda x. x \lambda x. x (\lambda x. x) \lambda x. x x \lambda x. x$

2. then we add parentheses around the abstractions and we get

$(\lambda x. x (\lambda x. x (\lambda x. x) (\lambda x. x x (\lambda x. x))))$

3. then, within the body of each abstraction, we apply left associative grouping and we treat everything between matching parentheses as one term

For the leftmost $\lambda x.$ there are two terms in the body of the abstraction, so there is no need to apply the left associativity rule

$(\lambda x. x (\lambda x. x (\lambda x. x) (\lambda x. x x (\lambda x. x))))$

For the next $\lambda x.$ there are 3 terms in the body of the abstraction and we group them as follows

$(\lambda x. x (\lambda x. ((x (\lambda x. x)) (\lambda x. x x (\lambda x. x)))))$

example continued on next page


Parsing general expressions (cont'd)

... example continued from previous page

$(\lambda x. x (\lambda x. ((x (\lambda x. x)) (\lambda x. x x (\lambda x. x)))))$

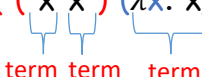
For the next $\lambda x.$ there is only one term and there is no need to add parentheses

$(\lambda x. x (\lambda x. ((x (\lambda x. x)) (\lambda x. x x (\lambda x. x)))))$



For the next $\lambda x.$ there are three terms and we group them as follows:

$(\lambda x. x (\lambda x. ((x (\lambda x. x)) (\lambda x. ((x x) (\lambda x. x)))))$



This is the end of the example.

More Examples

In the following examples, everything in black is in the original expression and everything in **red** is added by following the parsing rules

$(((a\ c)\ b)\ d)$

$(((\lambda x. x)\ (\lambda x. x)) (\lambda x. x))$

$((\ x\ x) (\lambda x. x))$

$((\ x\ x) (\lambda x. x))$

$(\overset{1}{\lambda x.} ((\overset{2}{x} (\overset{2}{\lambda x.} ((\overset{2}{x\ x}) x))) x) (\overset{3}{\lambda x.} ((\overset{3}{x\ x}) x)))$

Determining Bindings: Free and Bound Variables

After we parenthesize the expression, we determine free and bound variables.

x is **bound** to λx . if

--- same name ---▲

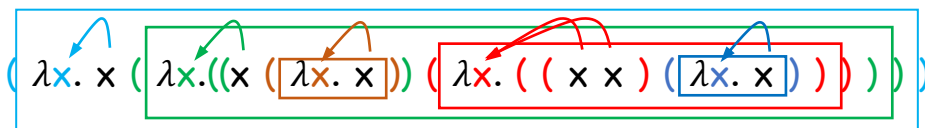
- x is in the body of the abstraction on λx . **and**
- λx . is the closest λx . to the left of x in whose body x appears

If x is not bound to any λx ., it is **free**.

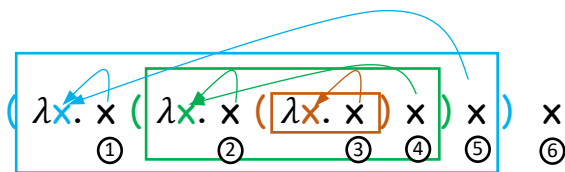
Another way to think about bound variables is the following: place a box around every abstraction. x is bound to λx . if

- x is in the box of the abstraction on λx . **and**
- The box of λx . is the **innermost** box containing x

Example 1:



Example 2:



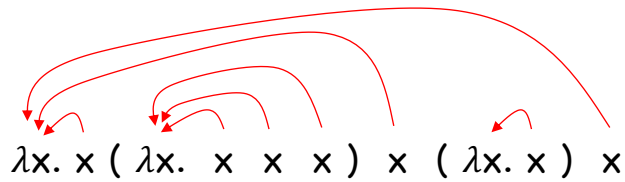
Some observations:

- ③ is not bound to λx . because the box of λx . is not the innermost box containing ③
- ④ is not bound to λx . because it is not in its box (body)
- ⑤ is not bound to λx . or λx . because it is not in their boxes (bodies)
- ⑥ is free because it is not in the box of any abstraction and therefore is not bound to any λx .

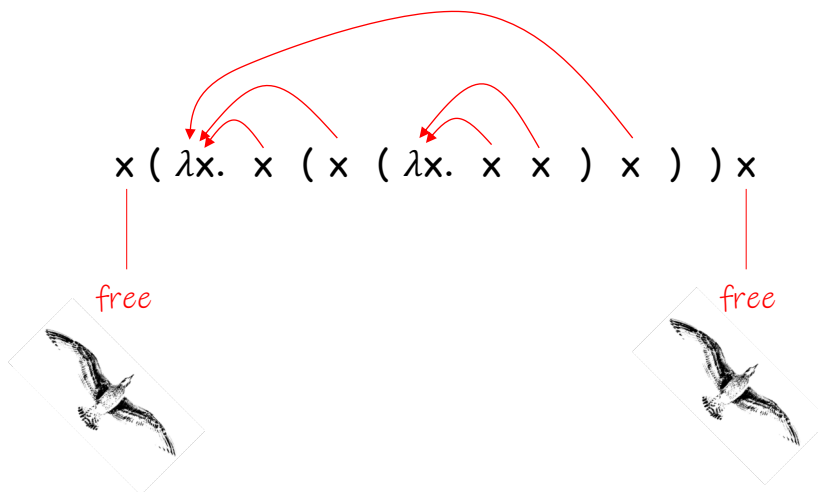
Determining Bindings: Free and Bound Variables

More Examples

Example 1:

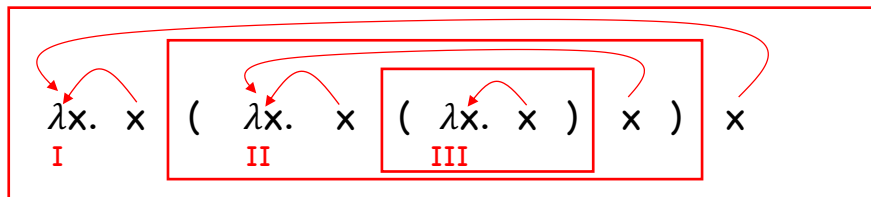


Example 2:

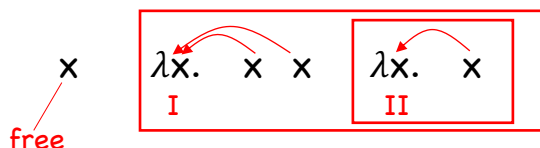


Determining Bindings: Free and Bound Variables

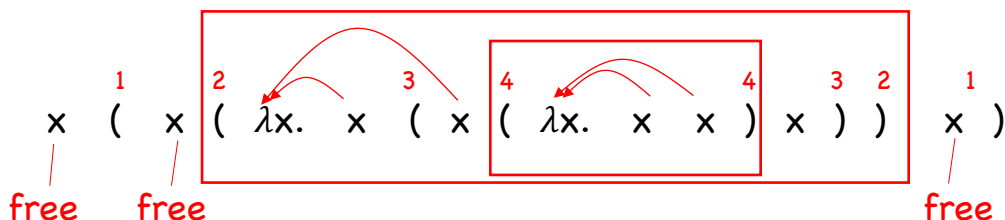
Example 3:



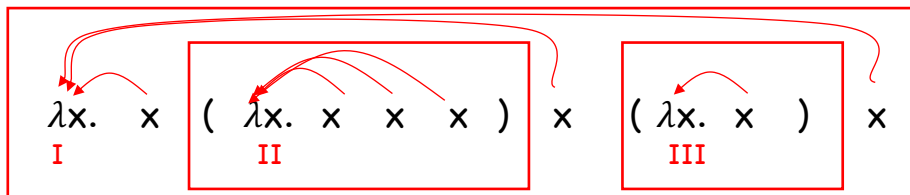
Example 4:



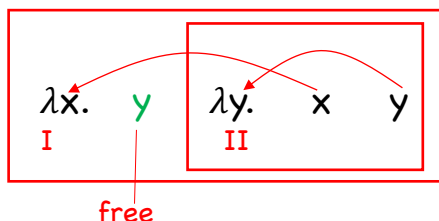
Example 5:



Example 6:

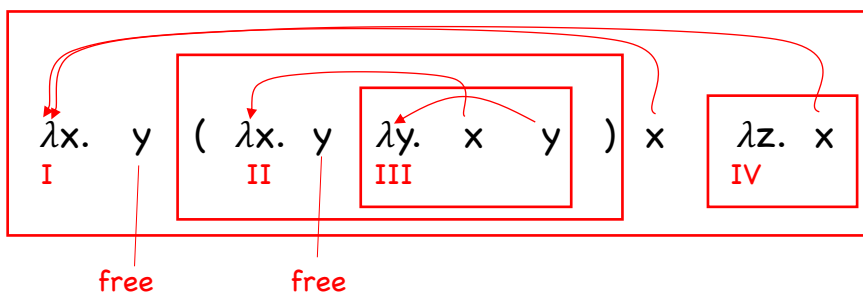


Example 7:



The first y is free because it is in the box of abstraction I on x . The name y is not the same as the name x , so y is not bound to λx .

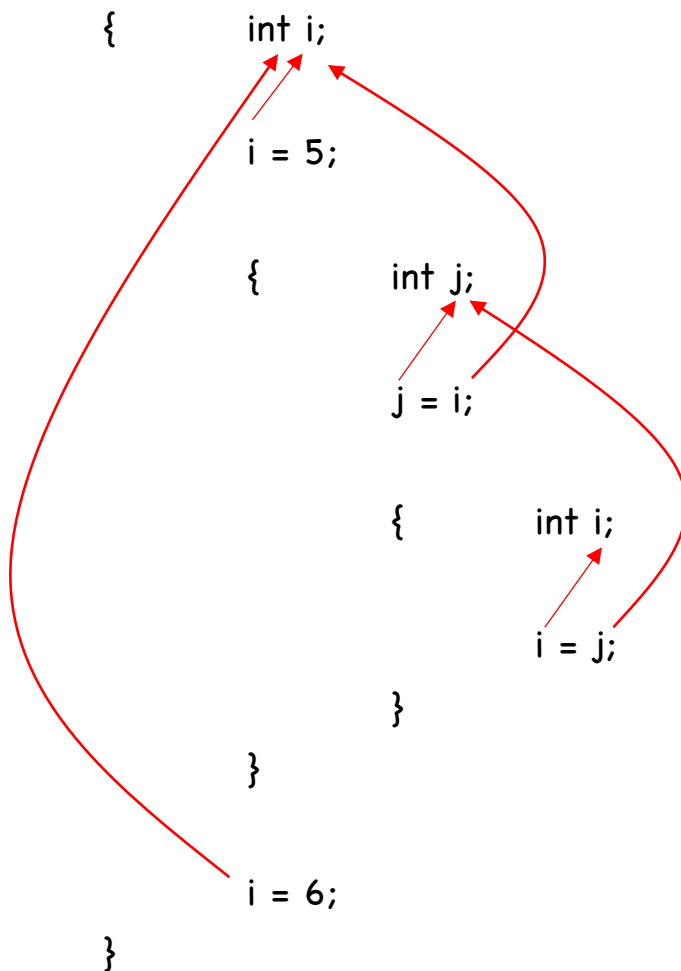
Example 8:



Note

The binding concept that we studied in lambda calculus is not that different from the concept of binding that you already know in a language like C (maybe you do not call it binding though).

You should be able to recognize how the binding we discussed is similar to the following example.



Recap

So far:

- identify bodies of abstractions (syntax)
- left associativity (syntax)
- parenthesizing (syntax)
- identify variables and their bindings (semantics)
- identify free variables (semantics)

Next:

- reducible expressions (redexes) (syntax)
- β -reductions (semantics)

Reducible Expressions

A reducible expression, also called a redex, is a **term** that has the form


$$(\lambda x. t) t'$$

where t and t' are terms

Remember. A reducible expression is a **term**. This means that you should be able to put parentheses around a reducible expression according to the rules for parenthesizing that we have seen.

To identify reducible expressions, you should start by parenthesizing everything
After parenthesizing, if there is a term of the form

$$((\lambda x. t) t')$$


term term

where t and t' are terms, then $((\lambda x. t) t')$ is a reducible expression

Note. This concept of grouping first before identifying reducible expressions is similar to how we evaluate arithmetic expressions.

$a + (b * c)$ these are correct parentheses added to the expression
we can evaluate the expression using this grouping

$(a + b) * c$ these are incorrect parentheses added to the expression
we cannot evaluate the expression using this grouping

Example of Reducible Expressions

1. $(\lambda x. (x\ x))$

has **no** redex because it will be grouped as shown by the added parentheses in red.

2. $x (\lambda x. x) x$

has **no redex** because it will be grouped as follows:

$$(x (\lambda x. x)) x$$

note that $(\lambda x. x) x$ are not grouped together and we cannot put parentheses around $(\lambda x. x) x$ so $(\lambda x. x)$ is not a redex

3. $x\ \lambda x. x\ \lambda x. x\ y\ z$

no redex

4. $\lambda x. \lambda y. x\ y\ z$

no redex

5. $(\lambda x. y) z$

has **one redex**. being a redex does not depend on the names of the variables

6. $\lambda x. y\ z$

no redex

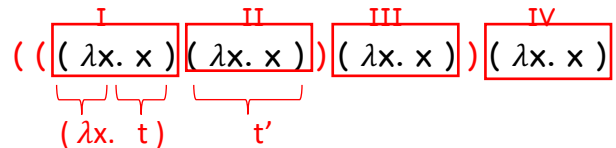
7. $(\lambda x. y) (\lambda x. y\ z (\lambda x. x))$

$(\lambda x. t) \quad t'$

one redex as shown (red parentheses are added and are not in the original expression)

Example of Reducible Expressions continued

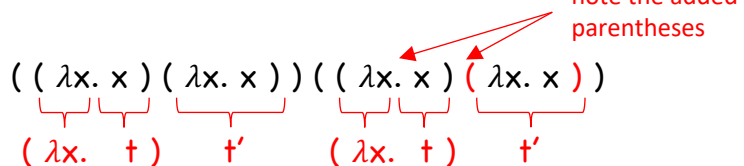
8. $(\lambda x. x) (\lambda x. x) (\lambda x. x) (\lambda x. x)$ has only one redex because it will be grouped as follows:



only I and II together form a redex as shown

note the added parentheses in red which prevent III and IV from being grouped together

9. $((\lambda x. x) (\lambda x. x)) ((\lambda x. x) \lambda x. x)$ has two redexes. They are the following



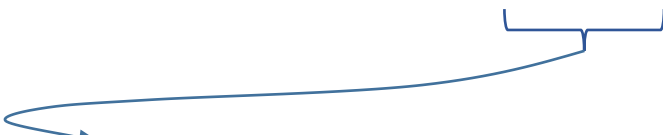
Note that there is no third redex here, a possibility that was raised in class. We cannot say that the third redex consists of two parts : $((\lambda x. x) (\lambda x. x))$ on the left and $((\lambda x. x) (\lambda x. x))$ on the right. The reason is that this would be missing a $\lambda x.$!!!

Definition: A lambda expression is in **normal form** if it contains no reducible expressions.

beta-reductions

β -reductions

After we have identified redexes, we can apply β -reductions

$$(\lambda x. t) t' \xrightarrow{\beta} [x \rightarrow (t')] t$$


should be read as: replace the x 's in t that are bound to λx . in the abstraction $(\lambda x. t)$, replace them with (t') and get rid of the λx . The results is t in which the replacement has been made.

Note that there are parentheses around t' . This is needed in general when t' is an abstraction.

Applying a function to an argument: You can think of reducing a redex as applying a function to an argument. the function is $(\lambda x. t)$ and the argument is t'

Renaming. If applying a β -reduction results in some variable changing its binding (free variable becoming bound or bound variable changes the abstraction to which it is bound), we need to do renaming first before applying the β -reduction. We discuss renaming later.

beta-reductions another explanation

β -reductions

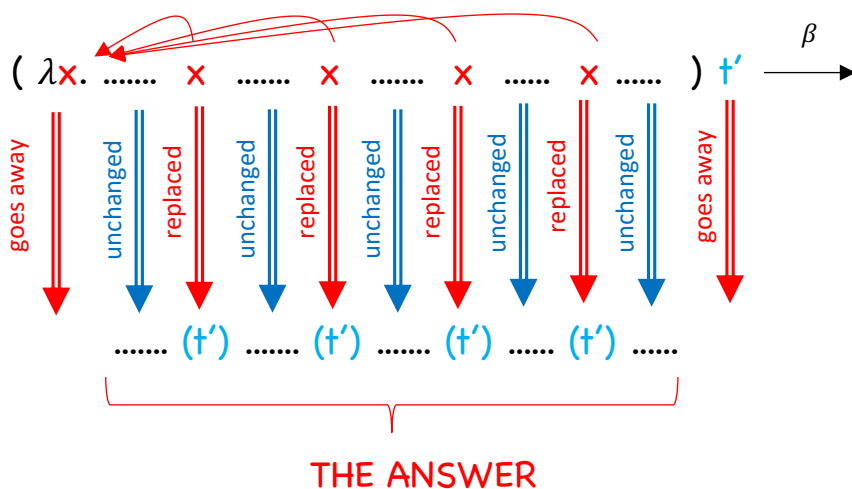
After we have identified redexes, we can apply β -reductions

$$(\lambda x. t) t' \xrightarrow{\beta} [x \rightarrow (t')] t$$

We can read this as follows.

replace all the x 's in t that are bound to the $\lambda x.$ of the redex, replace each one of them with (t') . Whatever remains of t after the replacement(s) is the answer.

This can also be illustrated as follows



So, the $\lambda x.$ goes away and each of the x 's that were bound to it get replaced with (t') and the t' (the argument) goes away

On the next couple pages, I give some examples

Examples of β -reductions

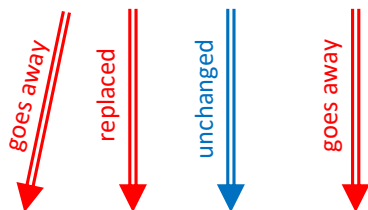
1. $(\lambda x. x y) (\lambda y. z)$

$$\underbrace{(\lambda x. x y)}_{\dagger} \underbrace{(\lambda y. z)}_{\dagger'} \xrightarrow{\beta} \underbrace{(\lambda y. z)}_{\dagger} \underbrace{y}_{\dagger'} \xrightarrow{\beta} z$$

This example deserves a little more explanation, especially that it is the first example we are looking at. Let us first look at the form of the expression

$(\lambda x. x y) (\lambda y. z)$ has the form

$(\lambda x. x \dots) (\lambda y. z)$ where the \dots stand for y , so we get



$(\lambda y. z) y$

In summary, the x gets replaced by $(\lambda y. z)$ and the y is left unchanged.

If we look at this resulting expression $(\lambda y. z) y$, it has the form

$(\lambda y. z) y$

$(\lambda y. \dots) y$

So, when we try to do the replacement, there is no y to replace and the \dots which is z is left unchanged and the argument y goes away, so we are left with z which is the answer as shown above.

Examples of β -reductions

$$2. (\lambda z. \lambda y. x z) (\lambda x. y) \not\rightarrow (\lambda \tilde{y}. x (\lambda x. y))$$

If we reduce this expression without renaming, y becomes bound to $\lambda y.$ to which it was not previously bound. We say that y is captured

before doing the replacement we should rename the $\lambda y.$ and any variables bound to it so that the substitution does not result in the capture of any variable. The name given to $\lambda y.$ should be a new name to avoid conflict with other variable names

The next two examples show how renaming is done

$$3. (\lambda y. \lambda z. y) z$$

before applying beta reduction, we should rename

$$(\lambda y. \lambda z. y) z \xrightarrow{\text{rename}} (\lambda y. \lambda w. y) z \xrightarrow{\beta} \lambda w. z$$

in the β -reduction step, we replaced the y in the body with z

Examples of β -reductions continued

$$4. \quad (\lambda y. \lambda z. y (\lambda x. x) z) z$$

Here also we should do renaming

$$\begin{aligned}
 (\lambda y. \lambda z. y (\lambda x. x) z) z &\xrightarrow{\text{rename}} (\lambda y. \lambda w. y (\lambda x. x) w) z \\
 &\xrightarrow{\beta} (\lambda w. z (\lambda x. x) w)
 \end{aligned}$$

We cannot reduce further because the grouping inside the abstraction is $(\lambda w. ((z (\lambda x. x)) w))$

$$5. \quad \lambda z. (\lambda y. \lambda z. y (\lambda x. x) z) z$$

Here also we have to do renaming because if we apply the reduction. if we do not rename before applying the reduction, z , which was bound by $\lambda z.$, becomes bound by $\lambda z.$

The general rules for renaming is the following:

1. if by doing beta-reduction a variable x that is bound to $\lambda x.$ becomes bound to another $\lambda x.$, we need to rename $\lambda x.$ by giving it a new name. Also, all the x 's that are already bound to the $\lambda x.$ should be given the new name.
2. If by doing beta-reduction a free variable x becomes bound to $\lambda x.$, we need to rename $\lambda x.$ by giving it a new name. Also, all the x 's that are already bound to the $\lambda x.$ should be given the new name.

Normal Order Evaluation

In normal order evaluation, leftmost outermost redexes are reduced first.

To reduce according to normal order, we need to:

1. identify redexes
2. determine redexes that are outermost
3. Amongst the redexes determined in step 2, reduce the leftmost one

This process repeats until there are no redexes left to reduce.

Let us look at the three steps above more closely

1. identify redexes: redexes have the form

$$(\lambda x. \textcolor{teal}{t}) \textcolor{teal}{t'}$$

$\textcolor{teal}{t'}$ is the right hand side of the redex.

$\textcolor{teal}{t}$ and $\textcolor{teal}{t'}$ are the terms of the redex

2. determine redexes that are outermost

A redex is nested inside another redex if it appears (or is equal to) one of its two terms.

Examples:

$$\frac{\frac{(\lambda \textcolor{teal}{x}. (\lambda \textcolor{red}{x}. \textcolor{red}{x}) \textcolor{teal}{x})}{(1)} \quad \frac{((\lambda \textcolor{teal}{x}. \textcolor{teal}{x}) \textcolor{teal}{x})}{(2)}}{(3)}$$

in this example redexes (1) and (2) are nested inside redex (3)

Normal order examples

$$\textcircled{1} \quad (\lambda x. (\lambda x. x) x) ((\lambda x. x) x)$$

Here we have three redexes as shown below

$$\frac{\frac{(\lambda x. (\lambda x. x) x)}{(1)} \quad \frac{((\lambda x. x) x)}{(2)}}{(3)}$$

redex (3) is reduced first because the other two redexes are nested inside it. We get

$$(\lambda x. (\lambda x. x) x) ((\lambda x. x) x) \rightarrow (\lambda x. x) ((\lambda x. x) x)$$

Now, we have two redexes

$$\frac{(\lambda x. x) \quad \frac{((\lambda x. x) x)}{(1)}}{(2)}$$

(1) is nested inside (2), so we reduce (2) first. We get

$$(\lambda x. x) ((\lambda x. x) x) \rightarrow ((\lambda x. x) x)$$

We are left with one redex. We reduce it to get

$$((\lambda x. x) x) \rightarrow x$$

Normal order examples

2

$(x \lambda x. ((\lambda x. x) x)) ((\lambda x. x) x)$

Here we have two redexes as shown below

$(x \lambda x. \underbrace{((\lambda x. x) x)}_{(1)}) \underbrace{((\lambda x. x) x)}_{(2)}$

Both redexes are outermost redexes because they are not nested inside other redexes, so we reduce redex (1) which is the leftmost outermost redex. We get:

$(x \lambda x. ((\lambda x. x) x)) ((\lambda x. x) x) \rightarrow (x \lambda x. x) ((\lambda x. x) x)$

Now, we have one redex remaining and we reduce it

$(x \lambda x. x) ((\lambda x. x) x) \rightarrow (x \lambda x. x) (x)$

There are no redexes remaining

Normal order examples

3

$((\lambda x. x)(\lambda x. x x)) ((\lambda x. x x) (\lambda x. x x))$

Here we have two redexes as shown below

$$\frac{((\lambda x. x) (\lambda x. x x))}{(1)} \quad \frac{((\lambda x. x x) (\lambda x. x x))}{(2)}$$

Both redexes are outermost redexes because they are not nested inside other redexes, so we reduce redex (1) which is the leftmost outermost redex

$$\begin{aligned} & ((\lambda x. x) (\lambda x. x x)) ((\lambda x. x x) (\lambda x. x x)) \\ \rightarrow & (\lambda x. x x) ((\lambda x. x x) (\lambda x. x x)) \end{aligned}$$

Now, we have two redexes as shown below:

$$\frac{(\lambda x. x x) \quad \frac{((\lambda x. x x) (\lambda x. x x))}{(2)}}{(1)}$$

Here (2) is nested inside (1), so we reduce (1) first. We get

$$\begin{aligned} & (\lambda x. x x) ((\lambda x. x x) (\lambda x. x x)) \\ \rightarrow & ((\lambda x. x x) (\lambda x. x x)) ((\lambda x. x x) (\lambda x. x x)) \end{aligned}$$

This expression does not have a normal form and I will stop reduction here.

Call by Value

In call by value, no reductions can be done inside abstractions. A redex cannot be reduced unless its right hand side is a value.

We start by defining what a value is.

Definition (Value). A value is expression that has no redex or an expression in which all redexes are under abstractions.

To reduce according to call by value order, we need to:

1. identify redexes
2. identify redexes that are not inside abstractions
3. Amongst the redexes determined in step 2, reduce the one whose right hand side is a value

This process repeats until there are no redexes left to reduce.

Let us look at the three steps above more closely

1. identify redexes: redexes have the form

$$(\lambda x. t) t'$$

t' is the right hand side of the redex.
 t and t' are the terms of the redex

2. identify redexes that are not inside an abstraction

An abstraction has the form

$$\lambda x. t$$

t is called the term of the abstraction

A redex is inside an abstraction if it appears in the term of an abstraction

Examples:

$$\lambda x. \underbrace{(\lambda x. x) x}_{(1)} \underbrace{((\lambda x. x) x)}_{(2)}$$

(3)

- (1): inside abstraction
(2): right hand side is a value
(3): right hand side is not a value

We reduce (2) first

Call by value examples

1

$(\lambda x. (\lambda x. x) x) ((\lambda x. x) x)$

Here we have three redexes as shown below

$$\frac{\frac{(\lambda x. (\lambda x. x) x)}{(1)} \quad \frac{((\lambda x. x) x)}{(2)}}{(3)}$$

redex (2) is reduced first because redex (1) is inside an abstraction and the right hand side of redex (3) is not a value. We get

$$(\lambda x. (\lambda x. x) x) ((\lambda x. x) x) \rightarrow (\lambda x. (\lambda x. x) x) (x)$$

Now, we have two redexes

$$\frac{(\lambda x. (\lambda x. x) x)}{(1)} (x)}{(2)}$$

Redex (1) is inside an abstraction, so we reduce redex (2) first. We get

$$(\lambda x. (\lambda x. x) x) (x) \rightarrow ((\lambda x. x) x)$$

We have one redex remaining. It is not inside an abstraction and its right hand side is a value. We reduce it to get

$$((\lambda x. x) x) \rightarrow x$$

Call by value examples

2

$(x \lambda x. ((\lambda x. x) x)) ((\lambda x. x) x)$

Here we have two redexes as shown below

$(x \lambda x. \underbrace{((\lambda x. x) x)}_{(1)}) \underbrace{((\lambda x. x) x)}_{(2)}$

Redex (1) is inside an abstraction and redex (2) is not inside an abstraction, so we reduce (2)

$(x \lambda x. ((\lambda x. x) x)) ((\lambda x. x) x) \rightarrow (x \lambda x. ((\lambda x. x) x)) (x)$

Now, we have one redex remaining but it is inside an abstraction, so we stop the evaluation.

$(x \lambda x. \underbrace{((\lambda x. x) x)}_{\text{cannot be reduced because it is inside an abstraction}}) (x)$

Call by value examples

3

$((\lambda x. x) (\lambda x. x x)) ((\lambda x. x x) (\lambda x. x x))$

Here we have two redexes as shown below

$$\frac{((\lambda x. x) (\lambda x. x x))}{(1)} \quad \frac{((\lambda x. x x) (\lambda x. x x))}{(2)}$$

Both redexes are not inside abstractions and both redexes have right hand sides which are values, so we can reduce either one first. If we reduce (1) first, we get

$$\begin{aligned} & ((\lambda x. x) (\lambda x. x x)) ((\lambda x. x x) (\lambda x. x x)) \\ \rightarrow & (\lambda x. x x) ((\lambda x. x x) (\lambda x. x x)) \end{aligned}$$

Now, we have two redexes as shown below:

$$\frac{(\lambda x. x x) \quad \frac{((\lambda x. x x) (\lambda x. x x))}{(2)}}{(1)}$$

Both redexes are not inside abstractions, but the right hand side of redex (1) is not a value, so we should reduce (2) first

$$\begin{aligned} & (\lambda x. x x) ((\lambda x. x x) (\lambda x. x x)) \\ \rightarrow & (\lambda x. x x) ((\lambda x. x x) (\lambda x. x x)) \end{aligned}$$

This expression does not have a normal form and I will stop reduction here.