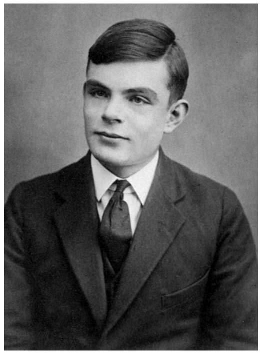
Lambda Calculus

Theoretical Foundations of Functional Programming

Raj Sunderraman
Computer Science Department
Georgia State University

Functions

- A computer program can be considered as a **function** from input values to output values. What does it mean for a function to be **computable**? The following 3 models are equivalent!
  - **Alonzo Church** defined Lambda Calculus in the 1930s to answer this question. He claimed that a function is computable if and only if it can be written as a λ-term.
  - **Alan Turing** devised Turing machines as a mechanism to define computability. He claimed that a function is computable if and only if it can be computed using a Turing machine.
  - **Kurt Gödel** introduced Recursive Function Theory to define computability. He claimed that a function is computable if and only if it is general recursive.



Lambda Calculus

- With its simple syntax and semantics, Lambda Calculus is an excellent vehicle to study the meaning of programming languages
- All functional programming languages (Haskel, LISP, Scheme, etc) are syntactic variations of the Lambda Calculus; so their semantics can be discussed in the context of Lambda Calculus
- Denotational Semantics, an important method for the formal specification of programming languages, grew out of Lambda Calculus

Three Observations About Functions

1. Functions need not be named

   x => x*x

2. The choice of name for the function parameter is irrelevant

   x => x*x

   y => y*y

   both are the same function (both return the square of their inputs)

3. Functions may be rewritten to have exactly one parameter

   (x,y) => x+y

   may be written as

   x => (y => x+y)

Concepts and Examples

Consider the function:

cube: Integer → Integer

where $cube(n) = n^3$

What is the value of the identifier "cube"?

How can we represent the object bound to "cube"?

Can we define this function without giving it a name? like a literal?

In Lambda Calculus, such a function would be represented by the expression:

$\lambda n.n^3$

This is an anonymous function (function literal) mapping its input n to n^3

Concepts and Examples

Consider another function:

$$f: \text{Integer} \times \text{Integer} \rightarrow \text{Integer}$$
$$\text{where } f(m,n) = n^2 + m$$

Lambda Calculus allows functions to have <u>exactly one parameter</u>

f would be represented by the expression:

$\lambda m.\lambda n.(n^2 + m)$

This is an anonymous function (function literal) mapping its input (m,n) to ($n^2 + m$) by "currying": m => (n => n^2 + m)

Lambda Calculus Syntax

A λ-term is defined inductively as follows:

1. A variable is a λ-term (e.g. x, y, m, n, etc)
2. If M is a λ-term and x is a variable, then (λx.M) is a λ-term
3. If M and N are λ-terms then (M N) is a λ-term

In the above definition,

(λx.M) is called a **lambda abstraction**; or in programming terminology the definition of a function. Here x is the input parameter (bound variable) and M is the body of the function.
(M N) is called a **function application**; or in programming terminology a function call. M is called the rator and N is called the rand (ope**rator**, ope**rand**)

Lambda Calculus Syntax continued

We introduce two other types of λ-terms:

4. A number is a λ-term (e.g. 10, 2, -5, 6.5, etc)

5. If M and N are λ-terms then (op M N) is a λ-term, where op is +, -, *, or /

These two are not part of the original "pure" Lambda Calculus.

Well-formed λ-terms:

x

5

(λx.x)

(λx.(* x x))

((λx.(* x x)) 5)

Parentheses; Lots of them!

(λx.(λy.(λz.((x z)(y z)))))

Let us see how this is constructed from the definition:

x, y, z are λ-terms using rule 1

(x z) is a λ-term using rule 3

(y z) is a λ-term using rule 3

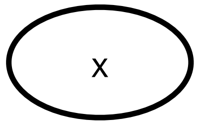
((x z) (y z)) is a λ-term using rule 3

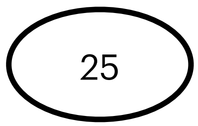
(λz.((xz)(y z))) is a λ-term using rule 2

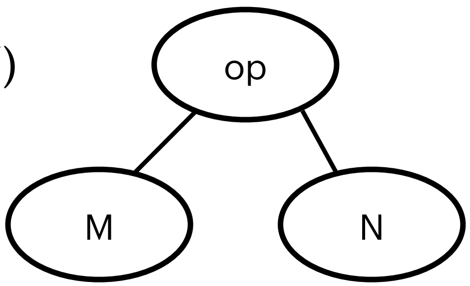
(λy.(λz.((x z)(y z)))) is a λ-term using rule 2

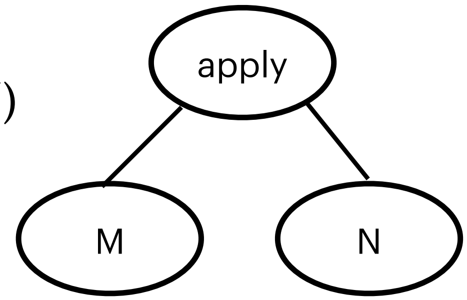
(λx.(λy.(λz.((x z)(y z))))) is a λ-term using rule 2

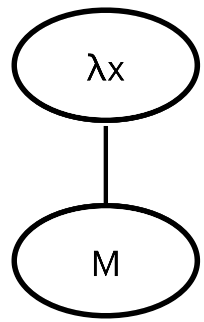
Expression Trees

Variable 

Number 

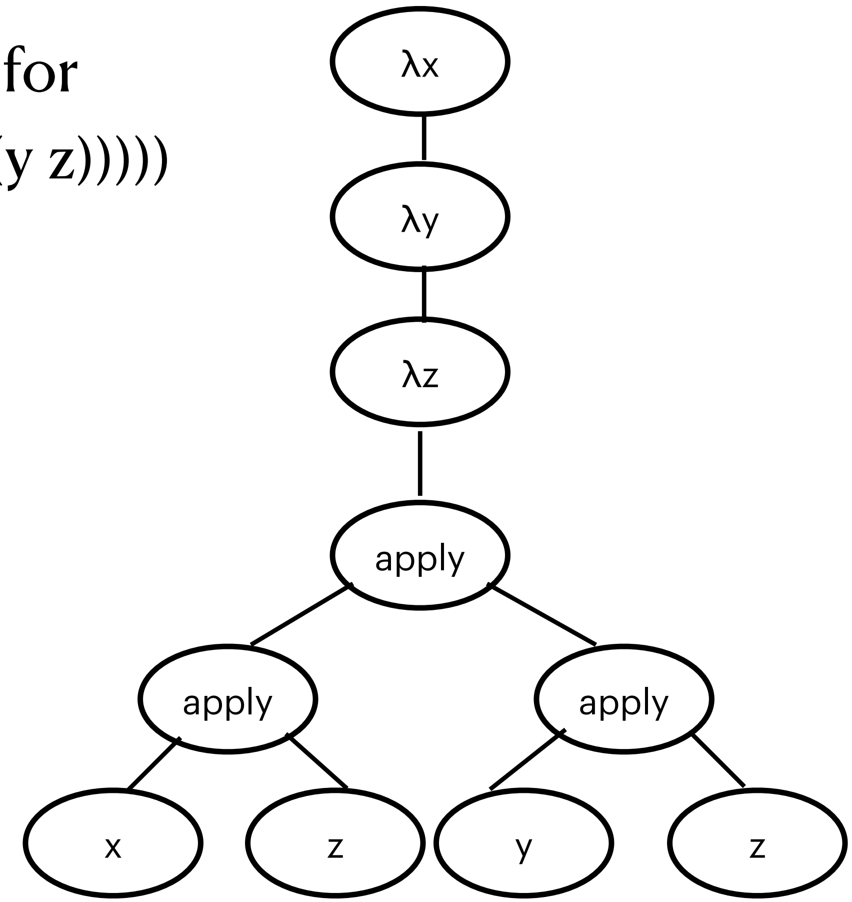
(op M N) 

(M N) 

(λx.M) 

Expression tree for

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



Conventions for omitting parentheses

1. Omit **outermost** parentheses. For example (λx.x) can be written as λx.x
2. Function **applications** are left-associative; So, omit parentheses when not necessary. For example (M N) P can be written as M N P
3. Body of function **abstractions** extend as far right as possible. So, we can write λx.(MN) as λx.MN

Using the above conventions, (λx.(λy.(λz.((x z)(y z))))) can be written as λx.λy.λz.x z (y z)

Lambda Calculus Interpreter (PLY Specification)

```
expr :
     NUMBER
   | NAME
   | LPAREN expr expr RPAREN
   | LPAREN LAMBDA NAME expr RPAREN
   | LPAREN OP expr expr RPAREN
```

```
NUMBER = r’[0-9]+ | [0-9]+”.”[0-9]* | “.”[0-9]*'
LPAREN = r’(‘
RPAREN = r’)'
OP = r’+|-|*|/‘
LAMBDA = r’[Ll][Aa][Mm][Bb][Dd][Aa]’
NAME = r‘[a-zA-z][a-zA-z0-9]*’
```

Lambda Calculus Interpreter continued

(λx.x) is written as (lambda x x)

(λx.(* x x)) is written as (lambda x (* x x))

((x y)(x z)) is written as ((x y)(x z))

The two syntactic differences are that

- the "." after λx is left out
- λ is spelt out as lambda

Lambda Calculus Semantics

What is the meaning (semantics, or value) of λ-terms?

e.g. what is the meaning of ((λx.(* x x)) 5) ?

Informally, it looks like we are calling the function (λx.(* x x)) with the argument 5. The function should return (* 5 5) = 25

Before we formally define the semantics of λ-terms , we need a few definitions.

- Free and Bound Variables
- α-equivalence
- Substitutions
- β-reductions

Free and Bound Variables

In the λ-term $(\lambda x.M)$

- x is a bound variable
- λ is said to **bind** x in M
- Any occurrence of x in M is said to be bound in $(\lambda x.M)$
- This concept is not novel! We have seen this in CSC 2510/Math 2420 in Predicate Calculus; e.g. in $\exists x\ P(x)$, x in $P(x)$ is bound to the x next to $\exists$.
- Also seen in programming languages such as Python in a formal parameter of a function (the occurrence of x in the function body is bound to the parameter x)

```
def f(x):
  return x*x
```

Free and Bound Variables - Examples

(1) In the λ-term, $\lambda x.\ x\ y$

- x next to λ is bound
- x in the body of the λ-term is bound to the x next to λ
- y in the body of the λ-term is free

(2) In the λ-term, $(\lambda \underset{b}{x}.\ \underset{b}{x}\ \underset{f}{y})(\lambda \underset{b}{y}.\ \underset{f}{z}\ \underset{b}{y})$

The variable next to λ is always bound!

(3) In the λ-term, $(\lambda x.(\lambda x.x)\ x)$, the x in the body of the inner λ-term is bound to the x of that λ-term and the last x is bound to the x of the outer λ-term.

Free Variable Definition

FV(M), the set of free variables in M is inductively defined as follows:

(1) $FV[x] = \{ x \}$

(2) $FV[\lambda x.M] = FV[M] - \{ x \}$

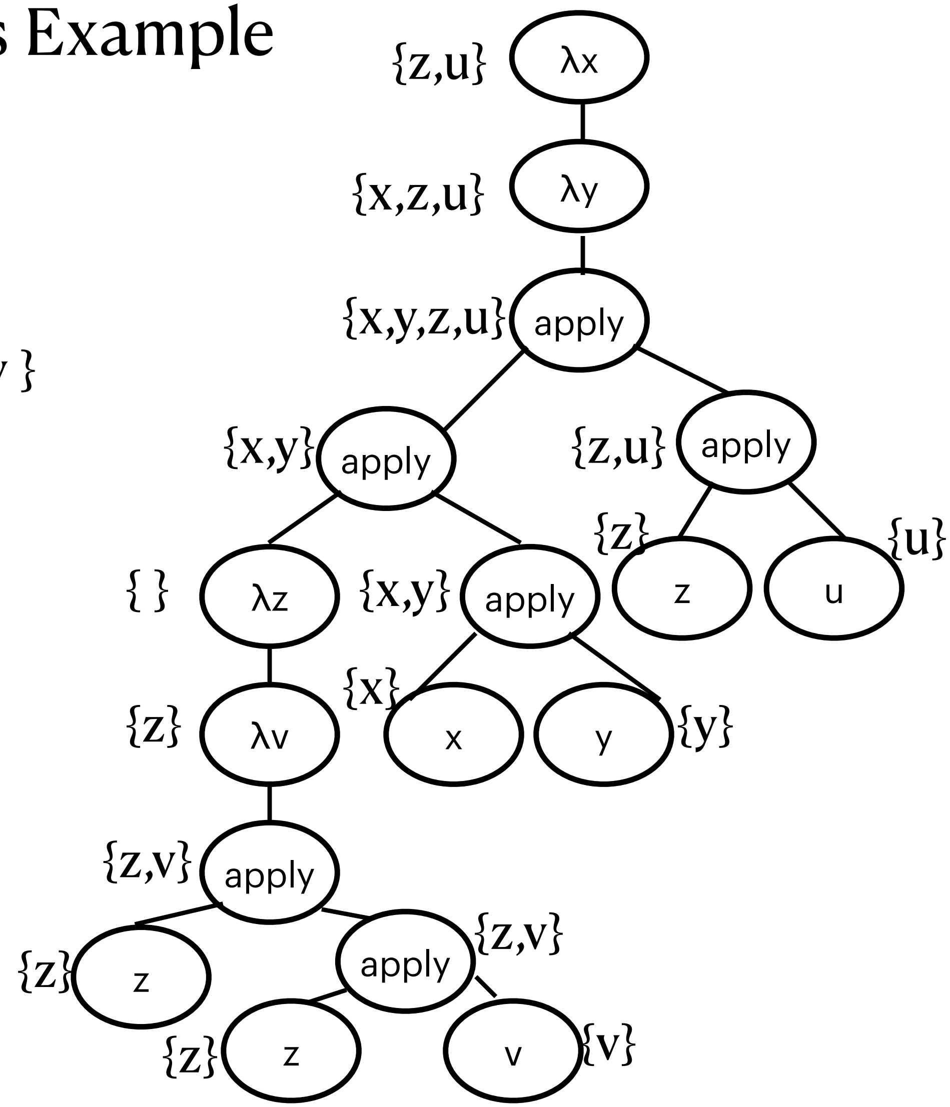
(3) $FV[MN] = FV[M] \cup FV[N]$

(4) $FV[number] = \{ \}$

(5) $FV[(op\ M\ N)] = FV[M] \cup FV[N]$

Free Variables Example

$$
\begin{aligned}
& FV[\lambda x.\lambda y.((\lambda z.\lambda v.z(zv))(xy)(zu))] \\
= {} & FV[((\lambda z.\lambda v.z(zv))(xy)(zu))] - \{ x, y \} \\
= {} & (FV[(\lambda z.\lambda v.z(zv))] \cup FV[(xy)] \cup FV[(zu)]) - \{ x, y \} \\
= {} & (FV[(\lambda z.\lambda v.z(zv))] \cup \{ x, y\} \cup \{ z, u \}) - \{ x, y \} \\
= {} & ((FV[z(zv)] - \{ z, v \}) \cup \{ x, y, z, u \}) - \{ x, y \} \\
= {} & ((\{ z, v \} - \{ z, v \}) \cup \{ x, y, z, u \}) - \{ x, y \} \\
= {} & \{ x, y, z, u \} - \{ x, y \} \\
= {} & \{ z, u \}
\end{aligned}
$$



α-equivalence

(λx.x) is the same as (λy.y)

(λx.(* x x)) is the same as (λu.(* u u))

All we have done is change the parameter name (**bound variable**) next to the λ as well as in the body of the function.

Renaming the bound variable does not change the abstraction.

Formally,

$(\lambda x.M) =_{\alpha} (\lambda y.M\{x \leftarrow y\})$

where y is a "brand new" variable not appearing in M, and
$M\{x \leftarrow y\}$ is M with all occurrences of x replaced by y.

α-equivalence continued

The same idea is present in programming languages as well. We do this often, i.e. we name a parameter of a function one way and after some time decide to give it a better name. To do this we consistently change all references to the old name with the new name!

e.g.

```
def isPrime(n):
 for i in range(1,n):
   if n%i == 0:
     return False
 return True
```

$=_\alpha$

```
def isPrime(num):
 for i in range(1,num):
   if num%i == 0:
     return False
 return True
```

Substitution

- Substitution is defined for **free** variables
- We will substitute a free variable with a λ-term.
- Substitution will be used during a "function call" when we provide an actual parameter value for the formal parameter
- For example, when we call the isPrime function with the actual argument 17, i.e. isPrime(17), the formal parameter n would have to be substituted by 17 in the body of the function:

```
def isPrime(n):
  for i in range(1, n):
    if n%i == 0:
      return False
  return True
```

isPrime(17) =

```
for i in range(1,17):
  if 17%i == 0:
    return False
return True
```

Substitution

$(\lambda x.\ (x\ y))\ [y = 5] \quad = \quad (\lambda x.\ (x\ 5))$
$(\lambda x.\ (x\ y))\ [y = (u\ v)] \quad = \quad (\lambda x.\ (x\ (u\ v)))$

Substitution must be done carefully so as not to alter the meaning of the λ-term!

$(\lambda x.\ (x\ y))\ [y = x] \quad \neq \quad (\lambda x.\ (x\ x))$

As can be seen, y was a free-variable before, but after the substitution y's value has become bound! Such a case is called a "**capture**" case.

$(\lambda x.\ (x\ y))\ [y = x] \quad =_\alpha \quad (\lambda x'.\ (x'\ y))\ [y = x] \quad = \quad (\lambda x'.\ (x'\ x))$

Another "capture" example:

$(\lambda x.\ (y\ x))\ [y = (\lambda z.(x\ z))] \quad \neq \quad (\lambda x.\ ((\lambda z.(x\ z))\ x))$
$(\lambda x.\ (y\ x))\ [y = (\lambda z.(x\ z))] \quad =_\alpha \quad (\lambda x'.\ (y\ x'))\ [y = (\lambda z.(x\ z))] \quad = \quad (\lambda x'.\ ((\lambda z.(x\ z))\ x'))$

Substitution Definition

1. $x\,[x = P]$	$=$	P	
2. $y\,[x = P]$	$=$	y	if $x \neq y$
3. $(M\ N)\,[x = P]$	$=$	$(M[x = P]\ \ N[x = P])$	
4. $(\lambda x.M)\,[x = P]$	$=$	$(\lambda x.M)$	
5. $(\lambda y.M)\,[x = P]$	$=$	$(\lambda y.M[x = P])$	if $x \neq y$ and $y \notin FV[P]$
6. $(\lambda y.M)\,[x = P]$	$=$	$(\lambda y'.(M\{y \leftarrow y'\}[x = P]))$	if $x \neq y$ and $y \in FV[P]$ and y' is brand new

Case 6 is the "capture" case! Bound variable y is "renamed" to y' using α-equivalence and then the substitution is applied.

Substitution Example

$(\lambda y. (((\lambda x. x)\ y)\ x))\ [x = (y\ (\lambda x. x))]$

$=$

$(\lambda y'. (((\lambda x. x)\ y')\ x))\ [x = (y\ (\lambda x. x))]$

$=$

$(\lambda y'. (((\lambda x. x)[x = (y\ (\lambda x. x))]\ y'[x = (y\ (\lambda x. x))])\ x[x = (y\ (\lambda x. x))]))$

$=$

$(\lambda y'. (((\lambda x. x)\ y')\ (y\ (\lambda x. x))))$

β-reduction

Consider the λ-term, (λx. (* x x)), that denotes the "square" function.

To call this function with argument 5, we will construct the "apply" λ-term:

((λx. (* x x)) 5)

β-reduction allows us to "execute" this function call. We "substitute" the bound variable (parameter), x, of the function abstraction with 5 in the body of the function abstraction.

((λx. (* x x)) 5) $=_\beta$ (* x x) [x = 5] = (* 5 5) = 25

β-reduction can be applied **only** to a λ-term of the form ((λx.M) N)

Note: The formal definition of substitution does not have rules for the impure λ-terms which involve arithmetic operators; but the definition can be easily extended.

β-reduction Definition

$((\lambda x.M)\ N)\ =_\beta\ M[x = N]$

A **β-redex** is of the form $((\lambda x.M)\ N)$

The result of β-reduction is called a **reduct**.

To "execute" a λ-term, β-reduction is applied repeatedly until there are no more β-redexes to be found in the λ-term.

A λ-term without any β-redexes is said to be in **β-normal-form.**

β-reduction Examples

$((\lambda x.y)\ (\lambda z.(z\ z))) =_\beta y[x = (\lambda z.(z\ z))] = y$

$((\lambda w.w)\ (\lambda w.w)) =_\beta w[w = (\lambda w.w)] = (\lambda w.w)$

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

$=_\beta ((\lambda x.y)\ ((z\ z)[z = (\lambda w.w)]))$

$= ((\lambda x.y)\ ((\lambda w.w)\ (\lambda w.w)))$

$=_\beta ((\lambda x.y)\ (w[w = (\lambda w.w)]))$

$= ((\lambda x.y)\ (\lambda w.w))$

$=_\beta (y[x = (\lambda w.w)])$

$= y$

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

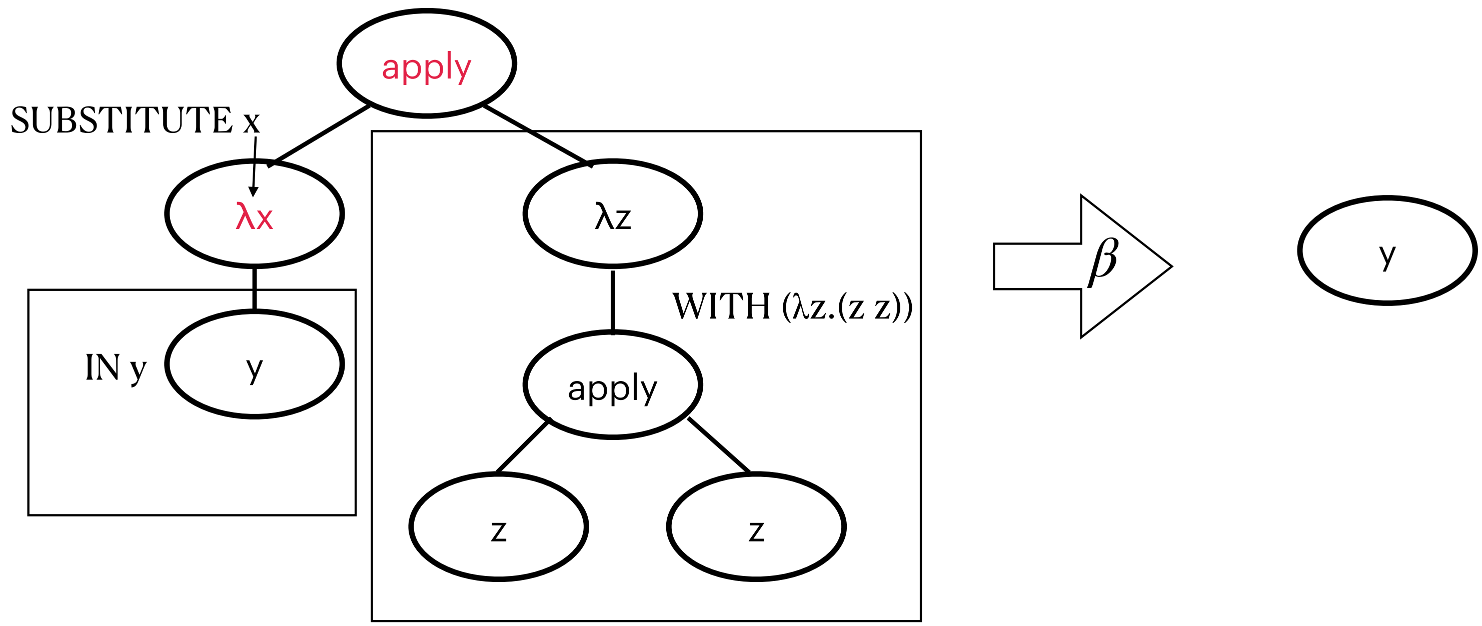
$=_\beta (y[x = ((\lambda z.(z\ z))\ (\lambda w.w))]$

$= y$

The order of applying β-reductions is not significant. The end result is the same, especially if it terminates.

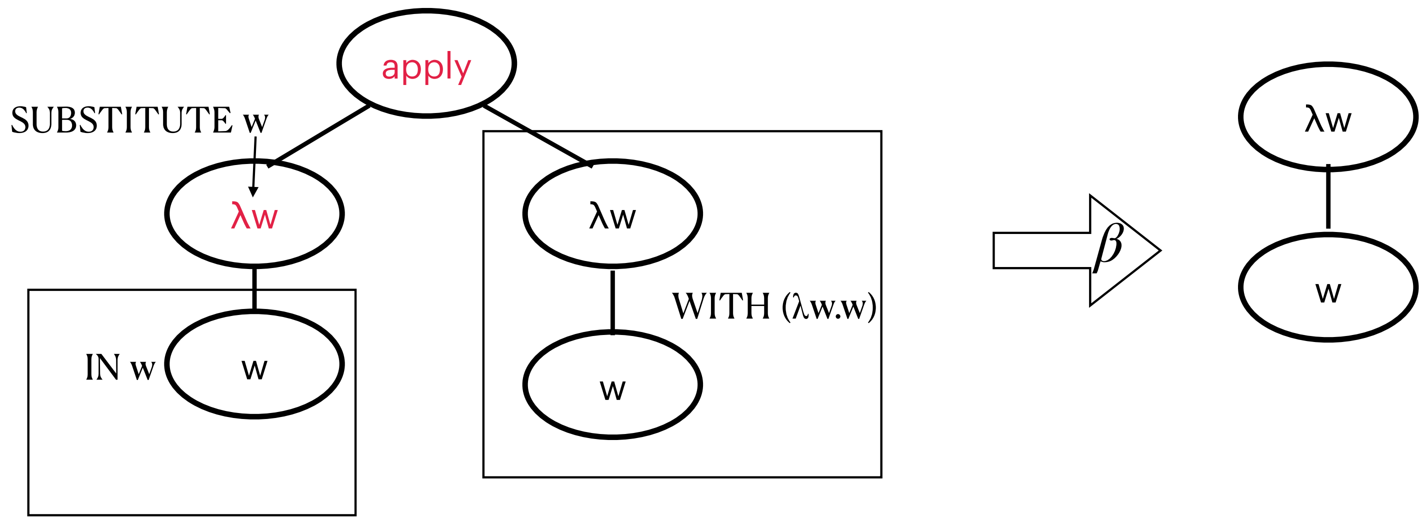
β-reduction Examples using Expression Trees

$((\lambda x.y)\ (\lambda z.(z\ z))) =_\beta y[x = (\lambda z.(z\ z))] = y$

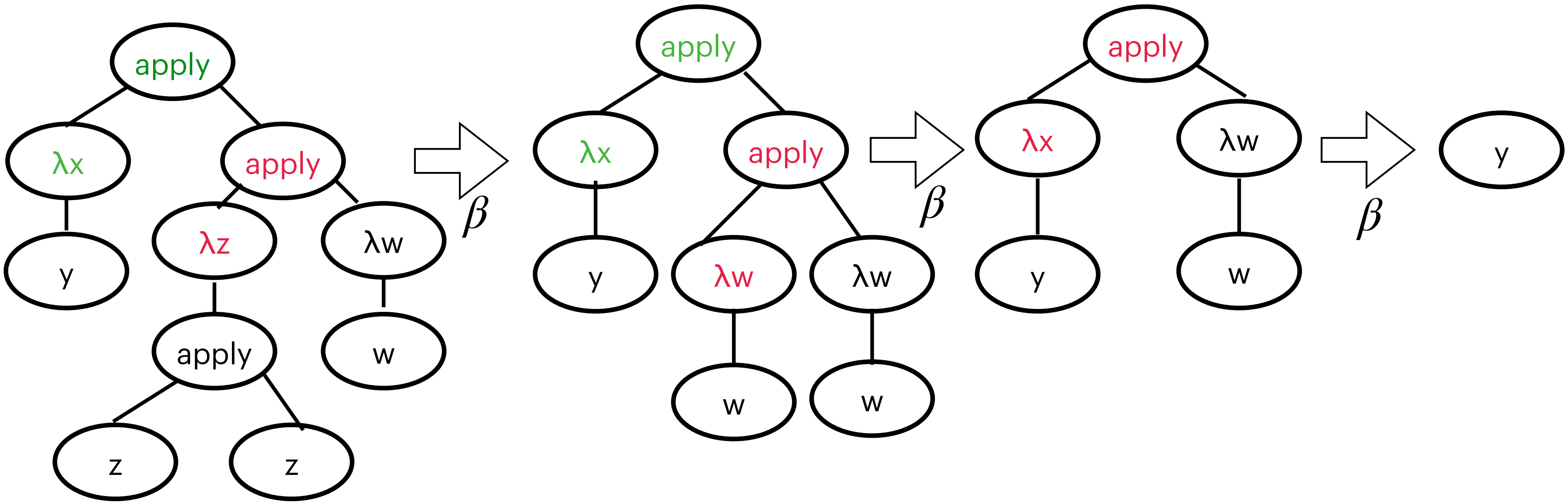


β-reduction Examples using Expression Trees

$((\lambda w.w)\ (\lambda w.w)) \ =_\beta \ w[w = (\lambda w.w)] \ = \ (\lambda w.w)$



β-reduction Examples using Expression Trees

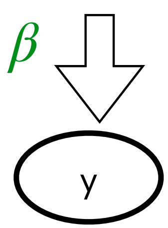


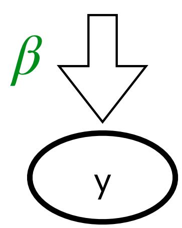
((λx.y) ((λz.(z z)) (λw.w)))

((λx.y) ((λw.w) (λw.w)))

((λx.y) (λw.w))

y



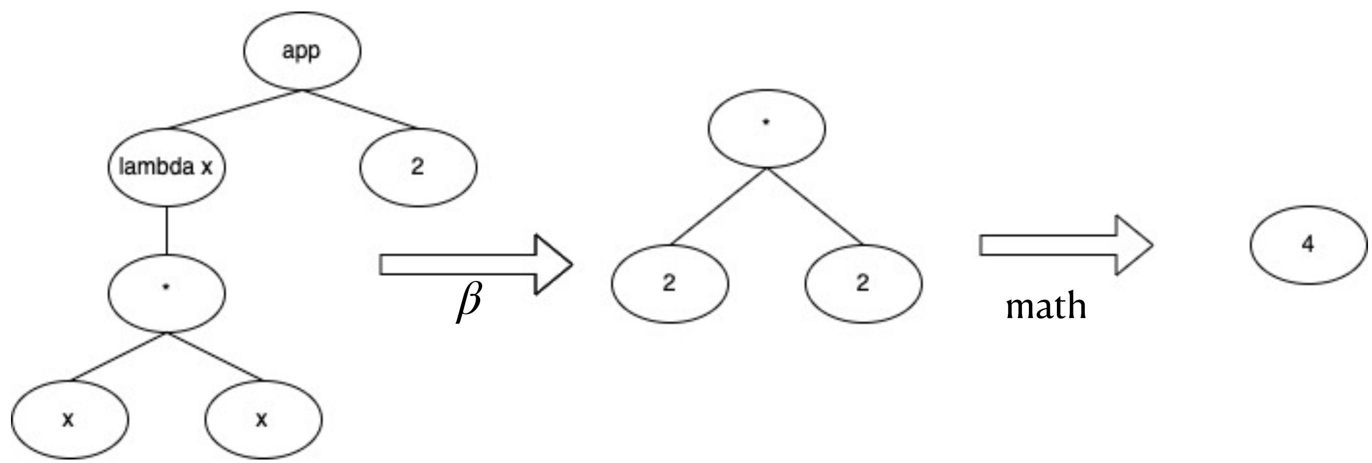


β-reduction Examples using Expression Trees

Using the Lambda Calculus Interpreter Notation:

((lambda x (* x x)) 2)

((lambda x (* x x)) 2)
$=_\beta$
(* 2 2)
=
4



β-reduction Examples using Expression Trees (HOF)

(((lambda f (lambda x (f (f x)))) (lambda y (* y (* y y)))) 2)

$=_\beta$

((lambda x ((lambda y (* y (* y y))) ((lambda y (* y (* y y))) x))) 2)

$=_\beta$

((lambda y (* y (* y y))) ((lambda y (* y (* y y))) 2)))

$=_\beta$

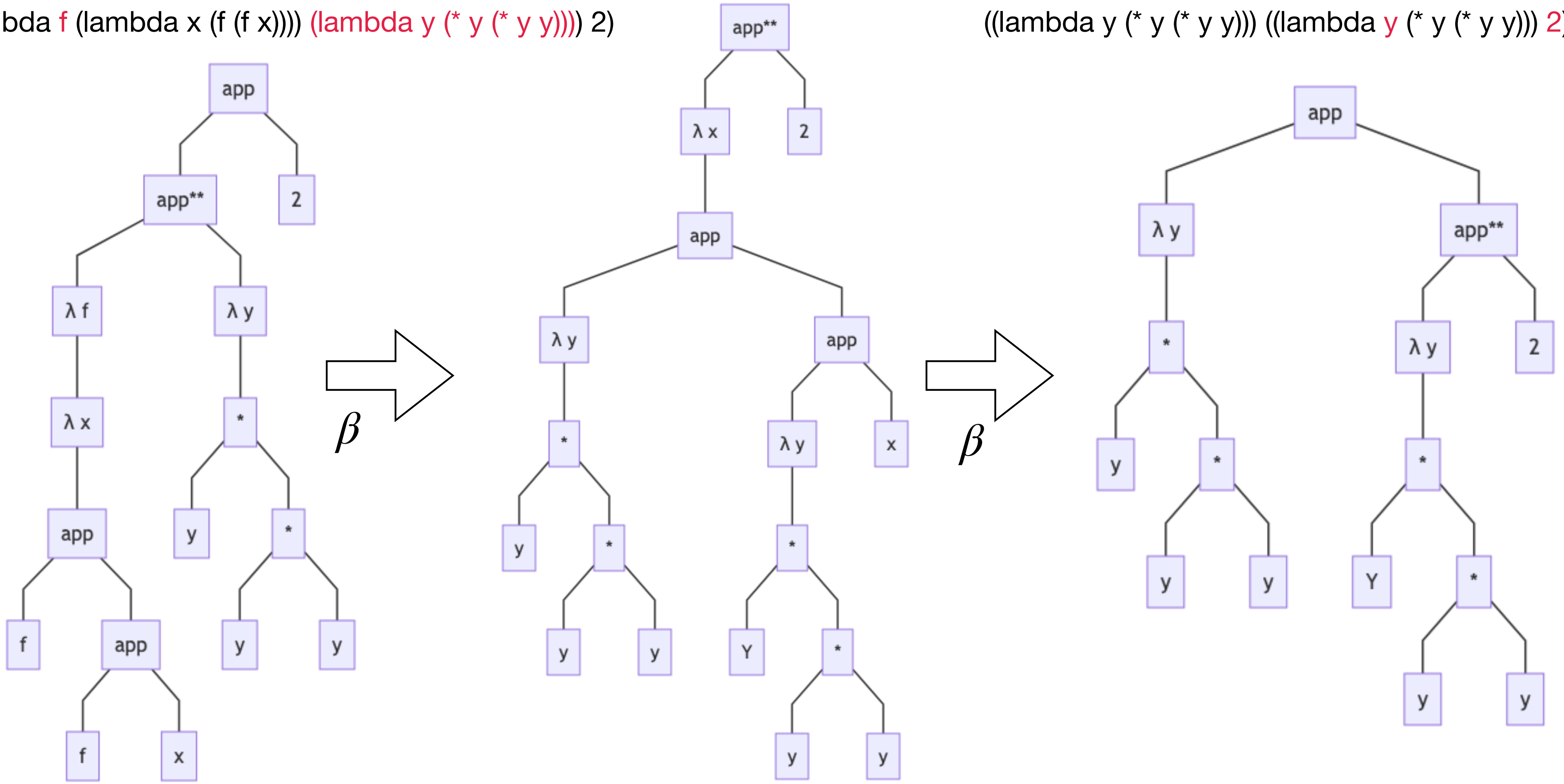
((lambda y (* y (* y y))) (* 2 (* 2 2))) = ((lambda y (* y (* y y))) 8)

$=_\beta$

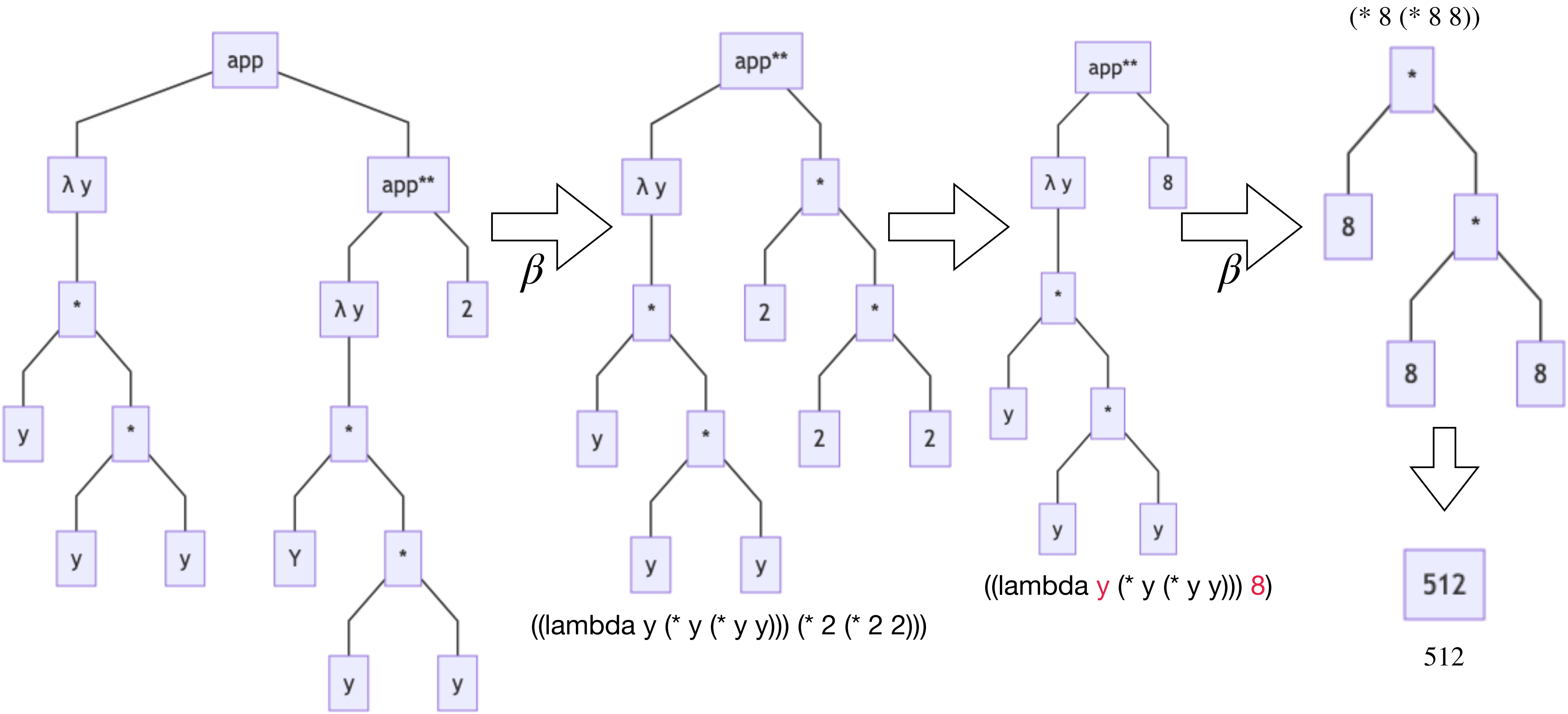
(* 8 (* 8 8)) = 512

(((lambda f (lambda x (f (f x)))) (lambda y (* y (* y y)))) 2)

((lambda y (* y (* y y))) ((lambda y (* y (* y y))) 2)))



((lambda x ((lambda y (* y (* y y))) ((lambda y (* y (* y y))) x))) 2) ((lambda y (* y (* y y))) ((lambda y (* y (* y y))) 2)))



Try this out!

((((lambda x (lambda y (lambda z (* (x z)(y z))))) (lambda x (* x x))) (lambda x (+ x x))) 5)

see if you can evaluate this to 250?