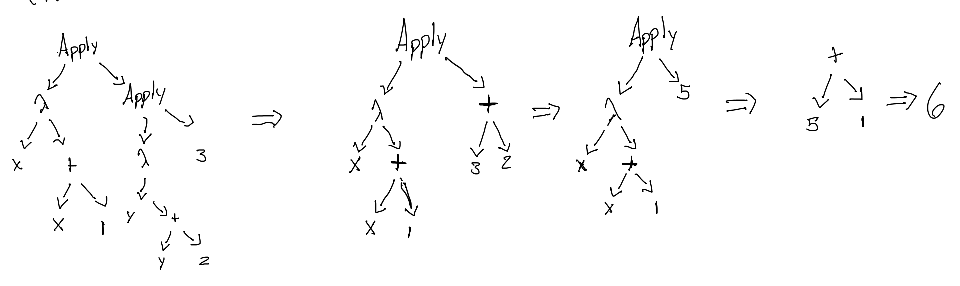
Lambda Calculus

Thursday, December 11, 2025 12:41 PM

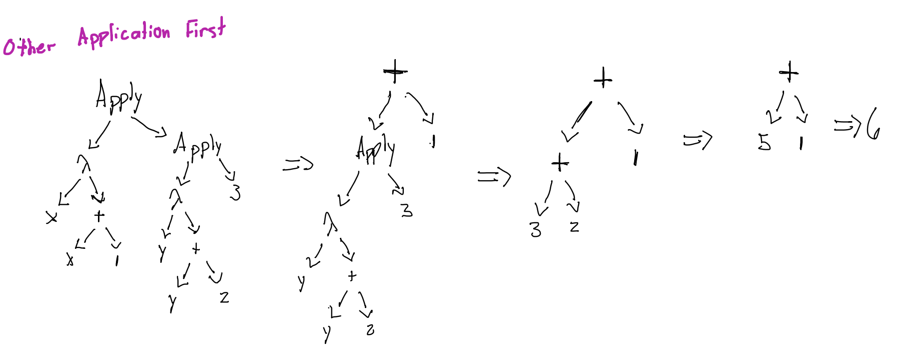
We write abstract syntax trees by finding applications of functions to arguments, and for each, replacing the formal param is the argument in function body. To do this, we must find an apply node whose left child is a lambda node, Since only lambda notes represent functions

- The right subtree of apply node is argument
- The left subtree of apply node (a lambda root) is function
- The left child of lambda is formal param.
- Right child of lambda is function body

$(\lambda x.x+1)((\lambda y.y+2)3)$



Other Application First



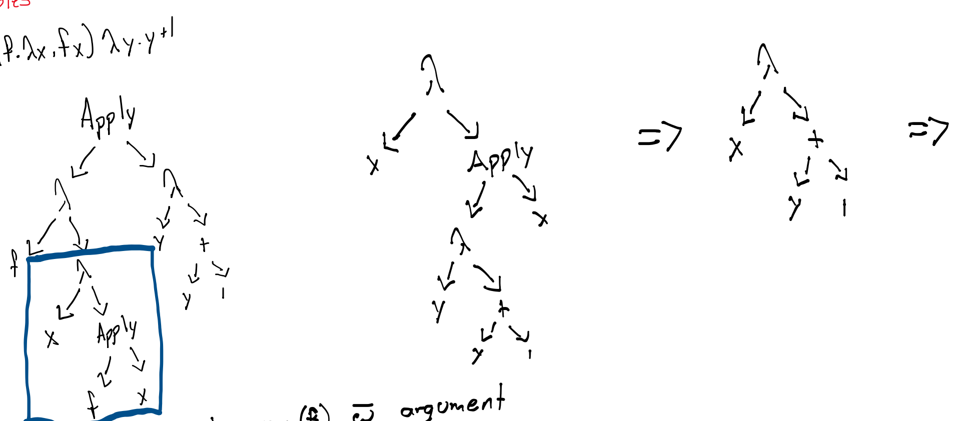
Examples

$(\lambda f.\lambda x.fx)\lambda y.y+1$

Outer lambda: $\lambda f.(\dots)$

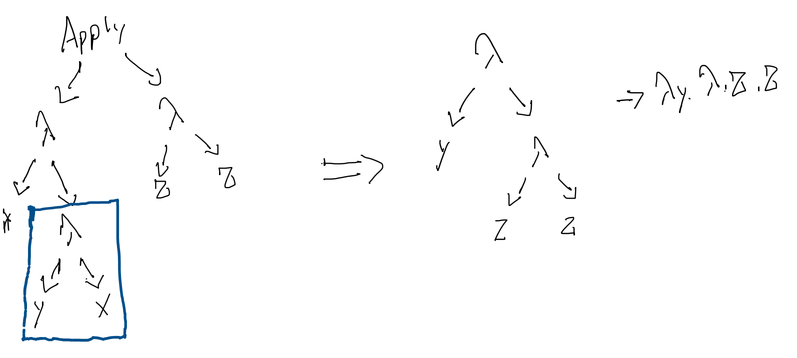
Argument to outer lambda: $\lambda y.y+1$

Body of outer lambda: $\lambda x.fx$

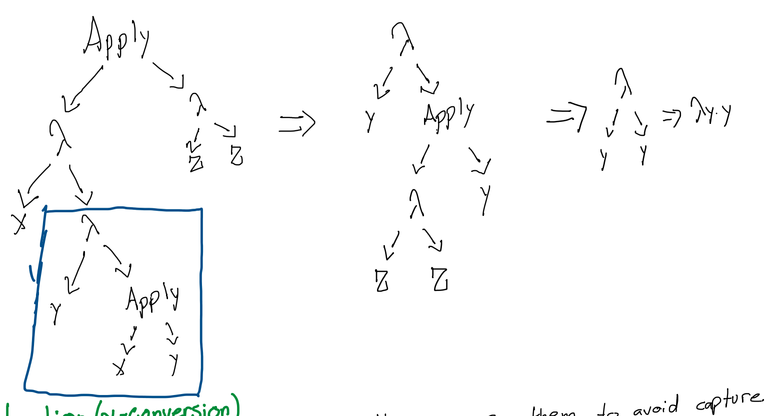


replace formal param (f) ω argument
$(\lambda y.y+1)$

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



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



Alpha Reduction (α-conversion)

Basic idea is formal parameters don't matter; rename them to avoid capture. Used to modify expressions of the form $\lambda x.M$. It renames all occurences of x that are free in M to some z that doesn't occur in M.

$\lambda x.x =_\alpha \lambda y.y$, $\lambda y.xy =_\alpha \lambda z.z$

Counter-example

$\lambda y.xy \neq_\alpha \lambda y.zy$ shouldn't rename free variables

$\lambda y.xy \neq_\alpha \lambda x.xx$ shouldn't conflict ω other variables

$\lambda x.E =_\alpha \lambda y.E[y/x]$ $(y \notin FV[E])$

Beta Reduction

$(\lambda x.M)N \to_\beta M[N/x]$

- left-hand side is **redex**
- right-hand side is **contractum**

This means M ω all free occurences of x replaced with N in a way that avoids capture.

We say that $(\lambda x.M)N$ beta reduces to M with N subbed for x.

Beta-Reduction of $((\lambda x.\lambda y.x)y)z$

$\to ((\lambda y.x)[y/x])z$ sub y for x in the body of $\lambda y.x$

$\to ((\lambda y'.x)[y/x])z$ after alpha reduction

$\to (\lambda y'.y)z$ first beta reduction complete

$\to y[z/y']$ sub z for y' in y

$\to y$ done!

η-reduction

Suppose x doesn't occur freely in f: $(\lambda x.fx)y \to fy$ (η-reduction)

- So $(\lambda x.fx)$ and f have the same behaviour

allows us to remove redondant lambdas

$\lambda x.fx \to f$, x doesn't occur freely

- f is a fixed function that doesn't depend on x
- $\lambda x.x+1$ does rely on x

Normal Form

There is another operation, beta expansion that we can also use.

We can beta expand $e_1 \to e_2$ iff e_2 Beta-reduces to e_1

Example

$(\lambda a.a)xy$

$(\lambda a.xy)(\lambda z.z)$

$(\lambda a.ay)x$

A computation is finished when there are no more redexes (no more applications of a function to an argument). A lambda expression without redexes is in normal form.

There is a normal form iff there is a sequence of beta reductions/expansions

β-normal form: expression without β-redices

- Doesn't always exist

Example of no β-normal form

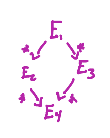
$(\lambda x.xx)(\lambda x.xx)$

$\to_\beta (xx)[(\lambda x.xx)/x]$

$= x[(\lambda x.xx)/x]\ x[(\lambda x.xx)/x]$

$= (\lambda x.xx)(\lambda x.xx)$

Church-Rosser Theorem



E_1, E_2, E_3, E_4 : λ-expression

$E_1 \to_\beta^* E_2$ and $E_1 \to^* E_3$

implies

there exists an E_4 such that $E_2 \to^* E_4$ and $E_3 \to E_4$

What does this mean?

λ-expression can reduce to at most one normal form

How do we choose redex towards the normal form?

Does the order matter?

- We have freedom to choose redices.
- Each prog. language clearly defines policy for computation
- Two typical ones:
  1) Normal Order Reduction (NOR)
  2) Applicative Order Reduction (AOR)

	Normal Order Reduction	Applicative Order Reduction
what to do	Always choose left-most and outermost redx. · outermost: not contained in any other redex	Always choose the leftmost and inner-most redex. · innermost: contains no other redex
What it implies	Arguments to a function won't be evaluated till needed.	Arguments to a function is evaluated before it's subsituted

Note: If an expression has a normal form, then NOR will find it.
Not true for AOR

Example: $(\lambda x.y)((\lambda x.xx)(\lambda x.xx))$

- NOR immediately reduces to y
- AOR gets stuck in loop (tries to reduce inner which (shown in example above) has no normal form.

Example in Python:

```
def fifteen(x):
    return 15
def infinite(x)
    return infinite(x + 1)
print fifteen(10)
```

1. print(infinite(1))
2. print(fifteen(infinite(1)))

Name	Expression	
1. Identity function - id	$\lambda x.x$	$(\lambda x.x)M = M$ for any M
2. Selection - fst	$\lambda x.\lambda y.x$	
3. Selection - snd	$\lambda x.\lambda y.y$	

fst takes two arguments and returns the first (ignoring second)

$(\lambda x.\lambda y.x)MN = (\lambda y.M)N = M$

fst takes the first argument, and returns a function

The returned function takes the second argument and returns the first one

4. Constant function $K = \lambda x.\lambda y.x$
   - $\lambda x.0$ returns 0 no matter what argument is given
   - $\lambda x.1$ is a constant function that returns 1
   - K 0 is $\lambda x.0$
   - Kn is a const. func. that returns n: $\lambda x.n$
   - K and fst are the same expression
5. Composition - comp $\lambda f.\lambda g$
   - Comp f g is the composition $f \circ g$ (math)
   - comp f f is equivalent to twice f
6. Self application - sa $\lambda x.xx$

Church Encoding (Boolean)

true $\equiv \lambda x.\lambda y.x$ fst

false $\equiv \lambda x.\lambda y.y$ snd

if a then b else c $\equiv$ a b c

Examples

if true then b else c

>> $(\lambda x.\lambda y.x)$ b c

$\to (\lambda y.b)c$

$\to b$

if false then b else c

>> $(\lambda x.\lambda y.y)$ b c

$\to (\lambda y.y)c$

$\to c$

Not x = if x then false else true

So,

Name	Expression
not	$\lambda x.x$ false true

not true

>> $(\lambda x.x$ false true) true

$\to$ true false true

>> $(\lambda x.\lambda y.x)$ false true

$\to (\lambda y.$false) true

$\to$ false

And x y = if x then y else false $\to \lambda x.\lambda y.xy$ false

or x y = if x then true else y $\to \lambda x.\lambda y.x$ true y

Natural Numbers

0	0	one
1	next(0)	five(one)
2	next(next(0))	five(five(one))
3	next(next(next(0)))	five(five(five(one)))
n	next(...(next(0)...)	five(...five(one)...)

0	$\lambda f.\lambda y.y$
1	$\lambda f.\lambda y.f\,y$
2	$\lambda f.\lambda y.f(f\,y)$
3	$\lambda f.\lambda y.f(f(fy))$
n	$\lambda f.\lambda y.f(\dots(fy)\dots)$

Operations on Numerals

plus one $= \lambda z.\lambda f.\lambda y.f(zfy)$

plus one 0

>> $(\lambda z.\lambda f.\lambda y.f(zfy))(\lambda f.\lambda y.y)$

$\to \lambda f.\lambda y.f((\lambda f.\lambda y.y)fy)$

$\to \lambda f.\lambda y.f((\lambda y.y)y)$

$\to \lambda f.\lambda y.fy$ (encoding of 1)

IsZero : $\lambda z.z(\lambda y.\text{false})$ true

isZero 0

>> $\lambda z.z(\lambda y.\text{false})$true $(\lambda f.\lambda y.y)$

$\to (\lambda f.\lambda y.y)(\lambda y.\text{false})$true

$\to (\lambda y.y)$true

$\to$ true

Arithmetic

$+ = \lambda M.\lambda N.\lambda x.\lambda y.(Mx)((Nx)y)$

$* = \lambda M.\lambda N.\lambda x.(M(Nx))$

So,

$(+MN) = \lambda x.\lambda y.(Mx)((Nx)y)$

$(*MN) = \lambda x.(M(Nx))$