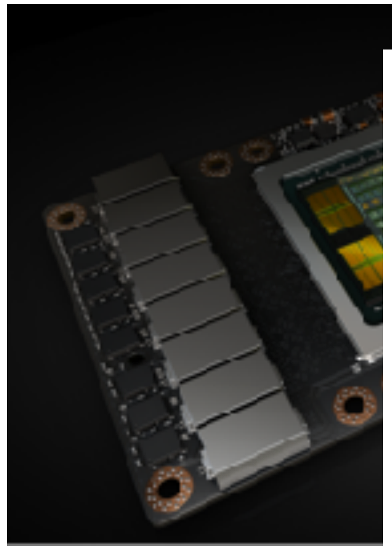
CS4111 - Computer Science

Lecture Set 3: Local and global variables

So far…

- Examples have been relatively straightforward
  - e.g. Only dealing with single function, no global variables
  - assuming single processing core
- From lecture 1:
  - *How do I design a program that can't be tested?*

Why this matters



Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	361,760	19,590.0	25,326.3	2,272
4	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7, Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
5	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890

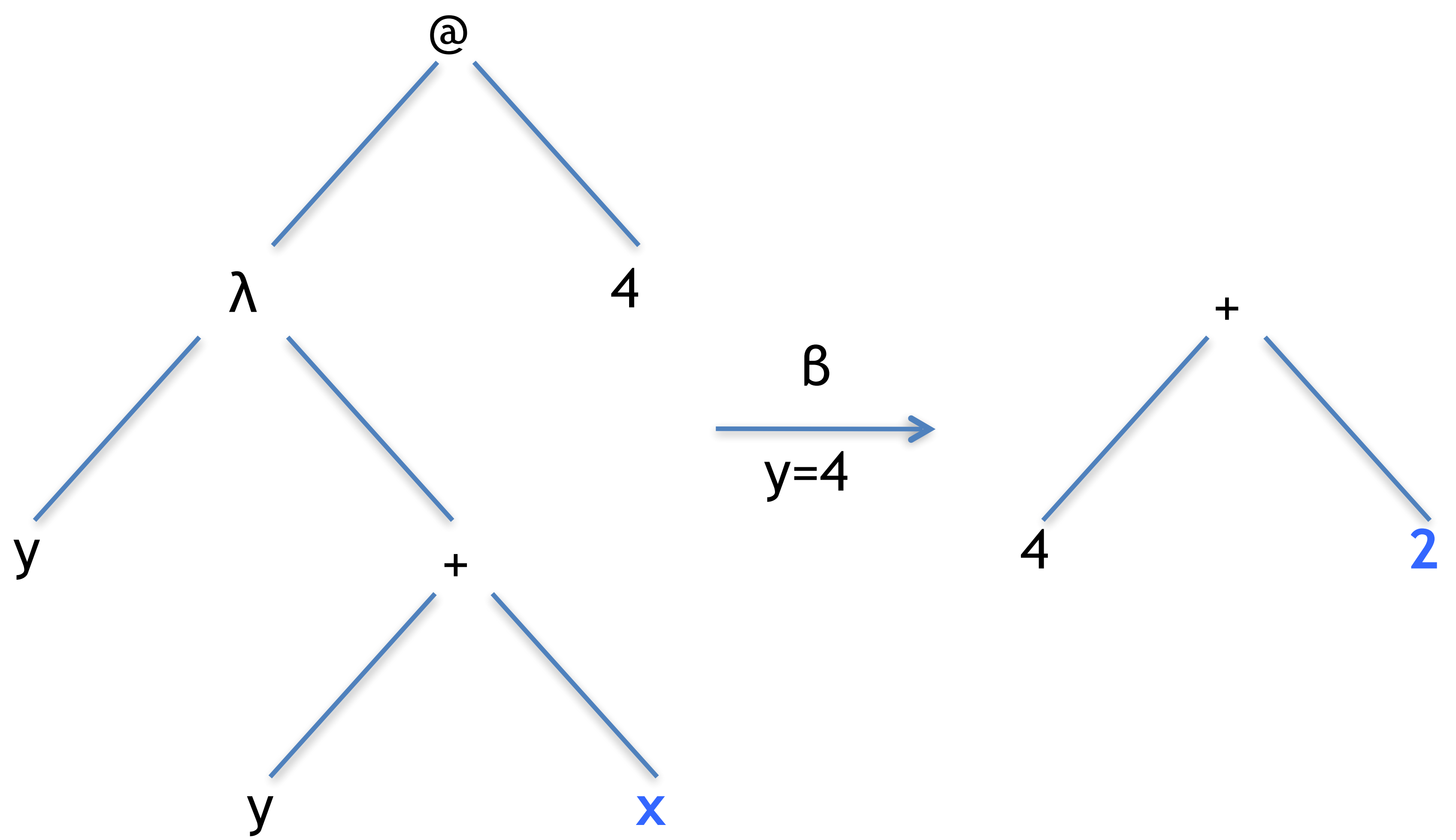
Necessary Tools

- Hundreds of functions
- Millions of copies of the **same** function running at the same time
- Functions taking other functions as parameters
- We need the ability to keep all these moving parts straight

Local and Global Variables

- > (define x 2)
- > (define addx (lambda (y) (+ y x)))
  – y is local, x is global
- > (addx 4)
- 6
- > x
- 2
- > (define x (addx 4))
- > x
- 6

Using ASTs



Local and global variables in λ calculus

- Some shorthand...
  - $\lambda x.\ E$
  - Function that takes one argument
  - Don't care what function does
- $\lambda x.\ (E\ F)$
  - Same as above, but two distinct parts to function
- Examples:
  - $\lambda x.\ +\ x\ y \equiv \lambda x.\ E$
  - $E = +\ x\ y$

- $\lambda x.\ + \ x\ y \equiv \lambda x.\ E\ F$
  - $E = +\ x$, $F = y$
  - OR, $E = +$, $F = x\ y$
- $\lambda x.\ x \equiv \lambda x.\ E$
- $E = x$
- But, $\lambda x.\ x <> \lambda x.\ E\ F$

- Terminology:
  - Global variable $\approx$ free variable
    - y is free in $(\lambda x.\ + x\ y)$
  - Local variable $\equiv$ bound variable
    - x is bound in $(\lambda x.\ + x\ y)$

Free Variables

- X is free in (E F) if X is free in E or in F.
- e.g from above, is y free in (+ x y)?
  - E = + x, F = y.
  - Not in E, but is in F.
  - It DOES occur free in (E F).
- Notice:
  - E = +, F = x y.
  - Not in E, but is in F.
  - It DOES occur free in (E F).

Bound Variables

- X is bound in (E F) if X is bound in E or in F.
- x occurs bound in (λ*y*. E) if
  - *x*=*y* AND *x* occurs free in E
  - OR, *x* is bound in E.
- Examples:
  - Is x bound in (λx. + x y)?
  - x is in the parameter list (λx) and it does appear free in (+x y)
  - Thus, x **<u>is</u>** bound in (<u>λx</u>. + x y).
  - **x** is a local variable in (λx. + **x** y).

Why two different letters?
To reflect generality

<u>NOTE</u>: Variable **x** is bound,
NOT the parameter **x**.

- Is y bound in (λx. + x y)?
  - It doesn't occur in parameter list
  - Other possibility? Bound in E?
    - E = + x y
  - Occurs free in E, so is NOT bound.
- More Examples:
  - e1: + x 3 x is free in e1
  - e2: (+ x) 3 (consider e2 = E F)
    - Free in E = (+ x), NOT free in F = 3
    - Therefore, free in e2

Question: How could y be bound in E?
Will address on next slide.

NOTE: Not Free ≠ Bound!
λx. + x 1
y is neither free nor bound.

- Is y bound in (λx. + (λy. + 3 y) x 2)?
  - It doesn't occur in parameter list
  - Other possibility? Bound in E?
    - E = (+ (λy. + 3 y) x 2)
    - It does appear in the parameter list
    - It **is** free in the body (+ 3 y)
      - Therefore, it is bound in E.
    - So, yes, y is bound in (λx. + (λy. + 3 y) x 2)

Why didn't we evaluate (λy. + 3 y) first?
Because it isn't a redex; it is missing an argument.
Will revisit order of evaluation soon.

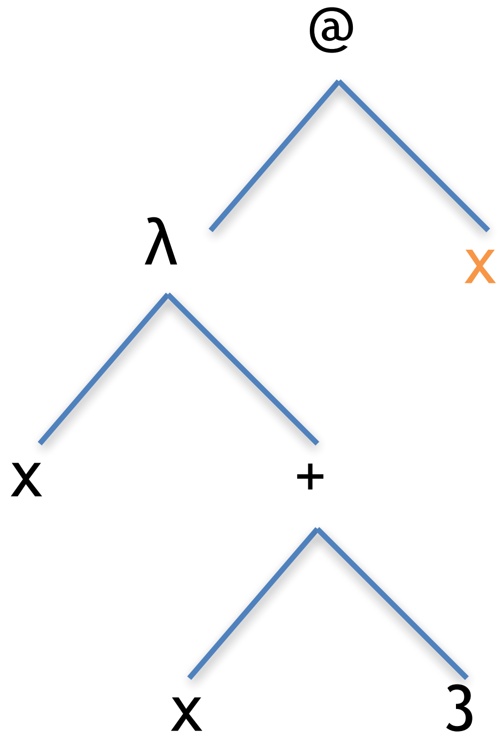
- Nested function
- ((λx. + (λy. + 3 y) x 2) 7)
- (+ (λy. + 3 y) 7 2)
- (+ (+ 3 7) 2)

- Different order of evaluation
- ((λx. + (λy. + 3 y) x 2) 7)
- ((λx. + (+ 3 x) 2) 7)
- (+ (+ 3 7) 2)

- (λx. + x 3) x

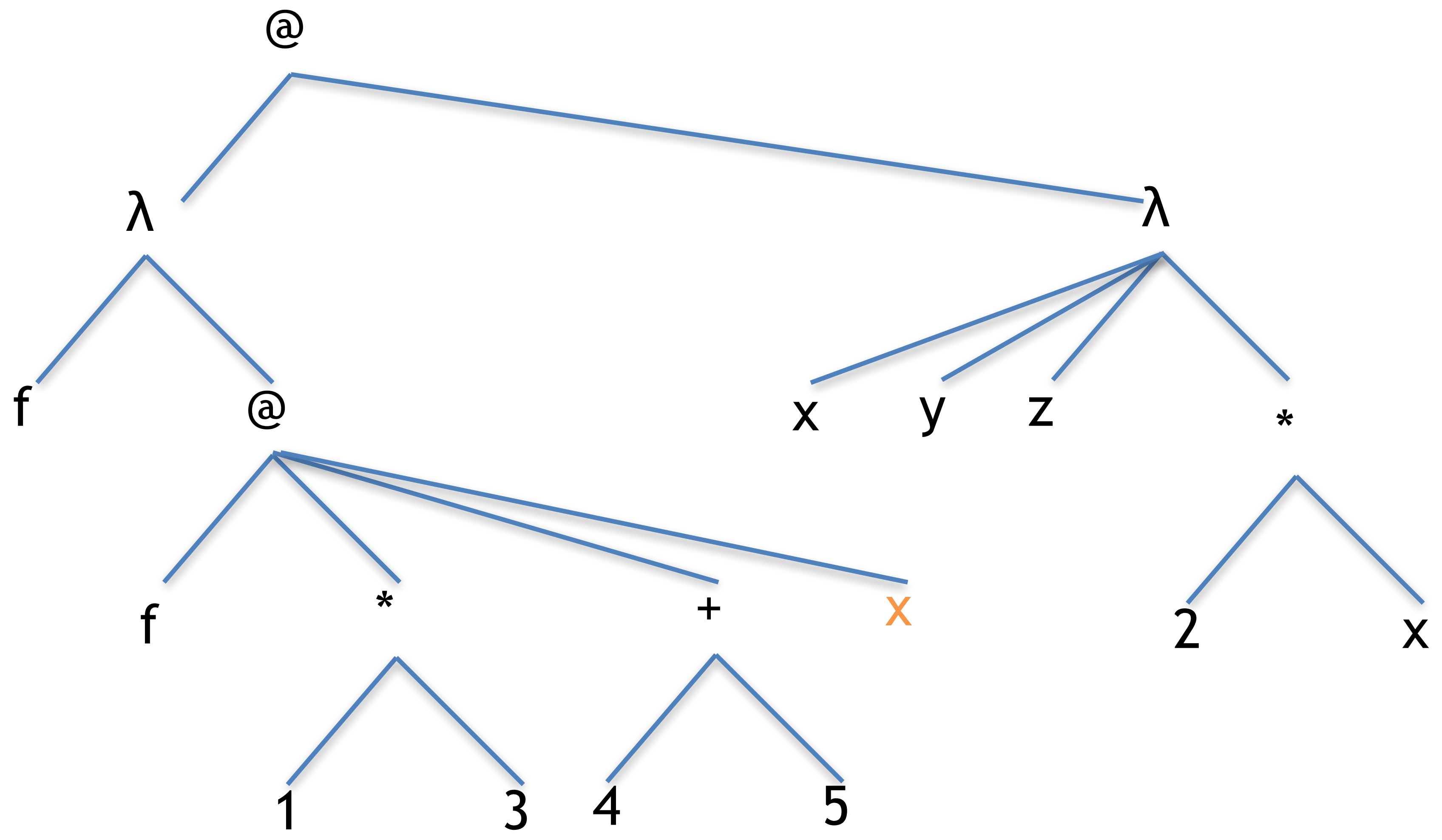
  E F

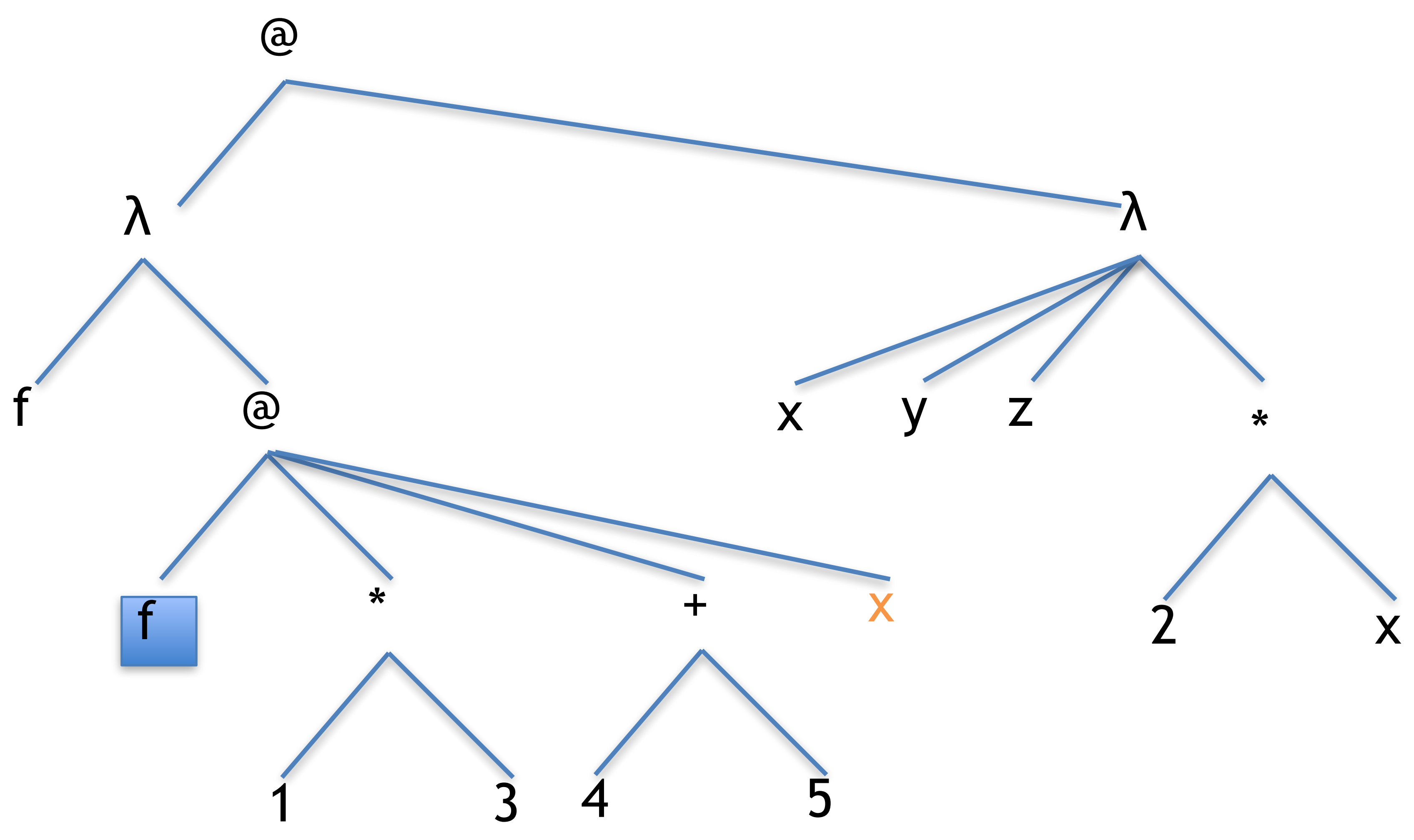
- x free or bound?
- Bound in E, free in F
- They are two different x's. The same name does **<u>not</u>** always mean same variable.

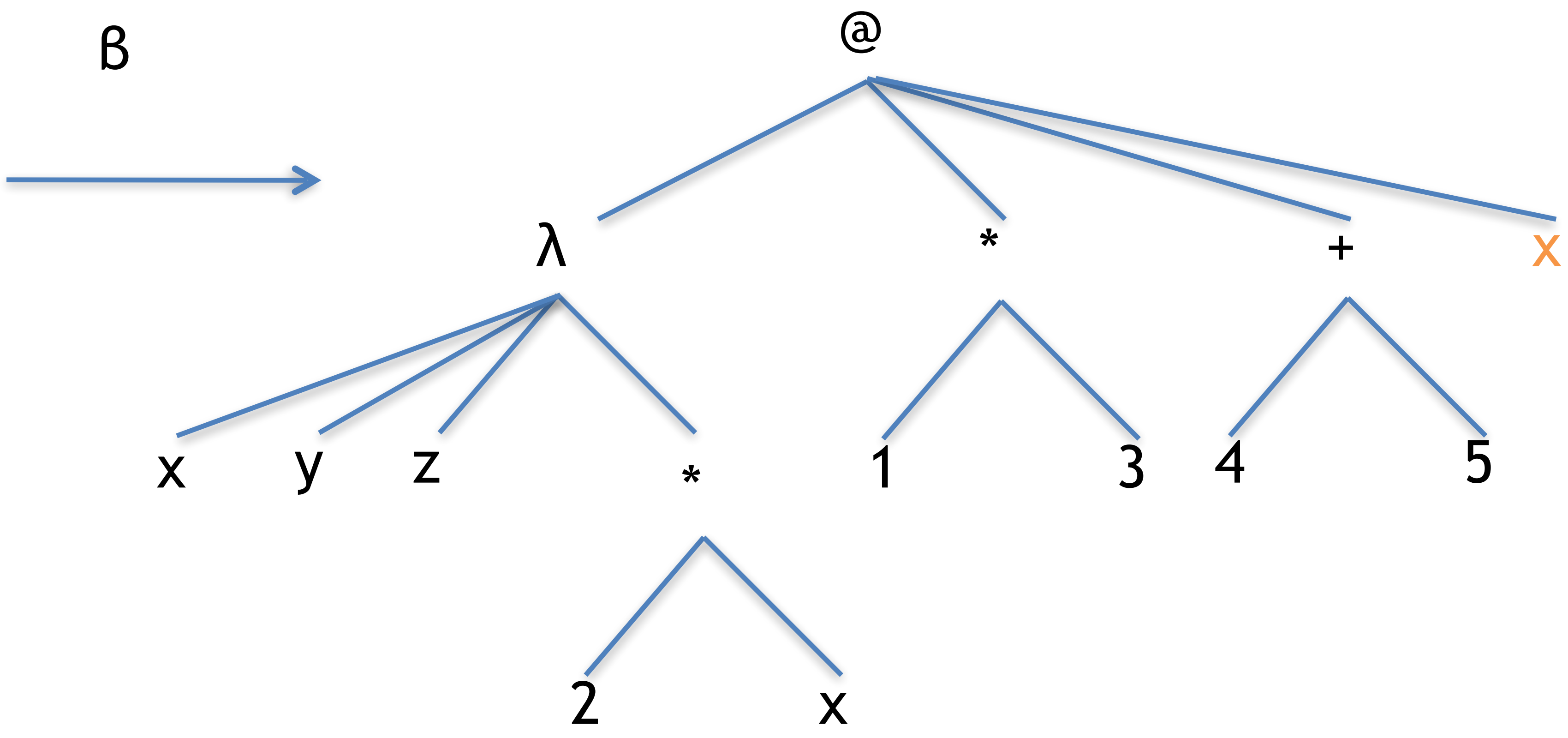


x has to have a value for this lambda to be a *redex*.
e.g.
(define x 4)

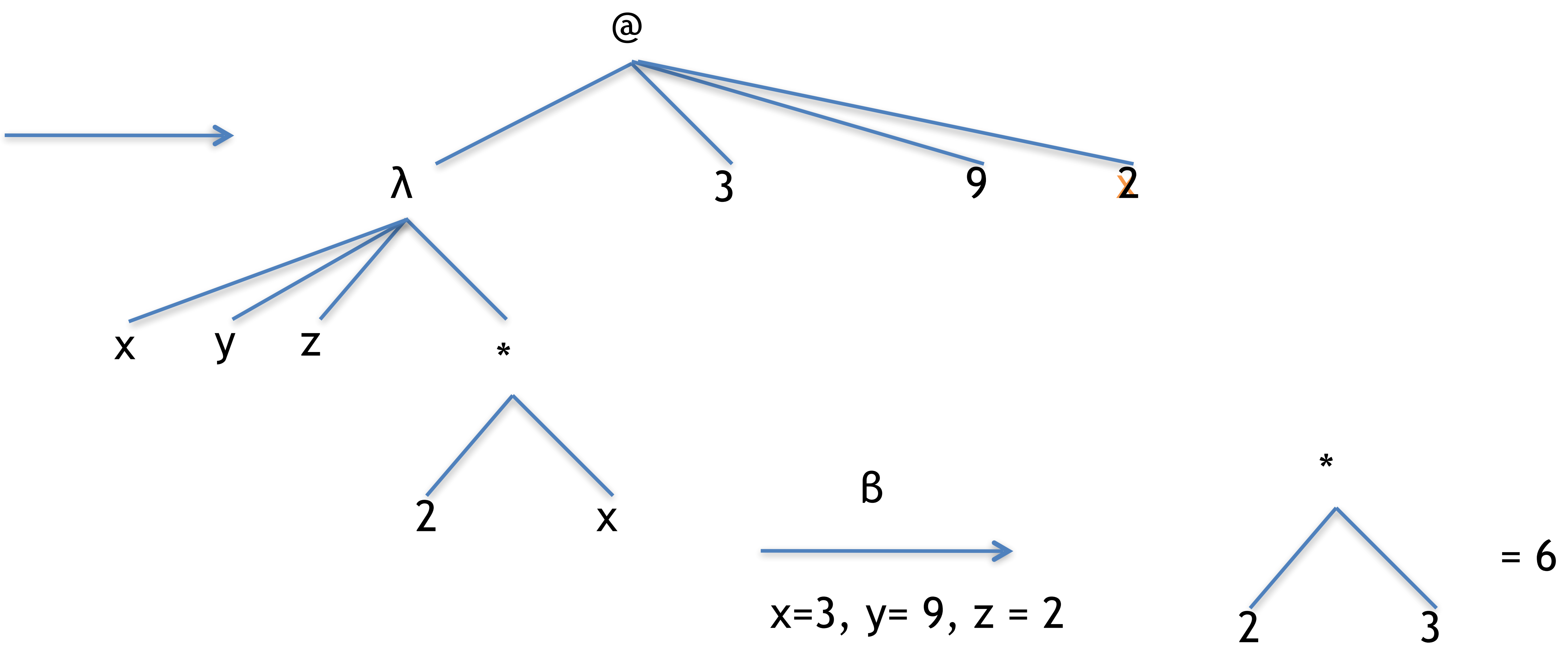
- (define x 2)
- ((λf. f (* 1 3) (+ 4 5) x) ((λxyz. (* 2 x)))
- Which x is free and which is bound?
- ((λf. f (* 1 3) (+ 4 5) x) ((λxyz. (* 2 x)))
  - x is free and x is bound.




@
λ
λ
f
@
x
y
z
*
f
*
+
x
2
x
1
3
4
5

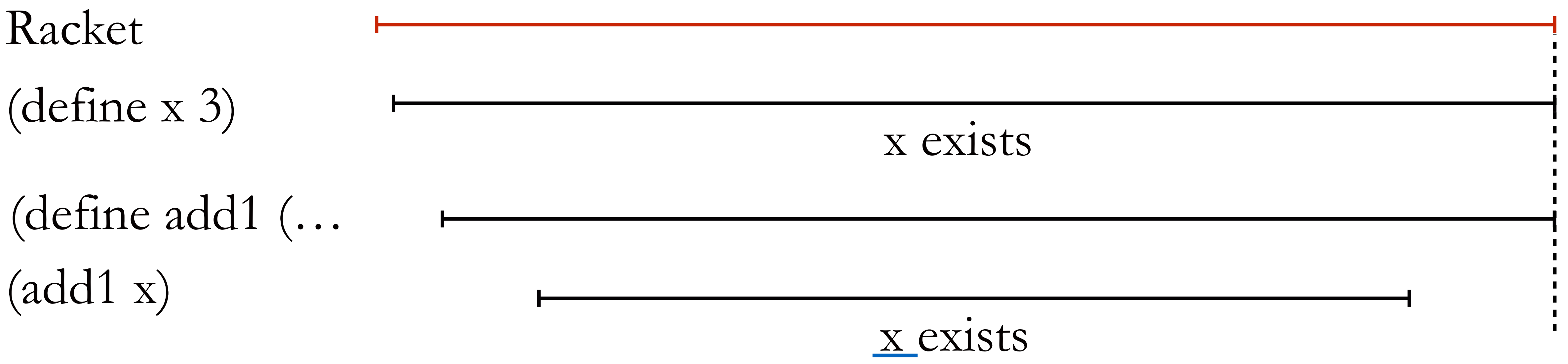

β
@
λ
*
+
x
x
y
z
*
1
3
4
5
2
x

(define x 2)



Recall lifetime diagrams (p. 14) xyz

```
> (define x 3)
> (define add1 (lambda (x) (+ x 1)))
> (add1 x)
```



Naming variables

- Variables with the same name are not necessarily the same variable
- Does not imply that two variables are the same
  – To avoid confusion better to keep different names

```
> (define x 3)
> (define add1 (lambda (x) (+ x 1)))
> (add1 x)
4
> x
3
```

Formally in λ calculus

- β reduction means passing arguments to a lambda.
- Remove the λ and parameters list (e.g. λxy.) and in the *resulting* body, replace the free variables with the arguments.
  - (λx. + x 1) 4
  - The body without λx. is (+ x 1)
  - In the *absence* of λx. part, x is free in (+ x 1)
  - Replace x with 4
  - => (+ 4 1) (redex)
  - =5
  - $(\lambda x.\ +\ x\ 1)\ 4 \xrightarrow{\beta} (+\ 4\ 1) = 5$

Nested functions

- ASTs/Prefix expressions can have multiple levels of nesting
  - E.g. (λx. * (+ 2 3) (- 2 (* 3 4))) 5
- But also:
  - (λx. + (λy. + 2 y) x 4) 3
  - Beta reduction replaces **free** occurrences in the body.
  - [x is free, *after* removing the (λx) part].
  - x=3 $\xrightarrow[\beta]{}$ + (λy. + 2 y) 3 4
  - y=3 $\xrightarrow[\beta]{}$ + (+ 2 3) 4
  - = 9

- A more confusing but *identical* example:
  - (λx. + (λx. + 2 x) x 4) 3
- Replace only FREE occurrences, after removing λx.
  - x=3 $\xrightarrow{\beta}$ + (λx. + 2 x) 3 4
  - Note x is not replaced, because it is ***still*** bound (to lambda starting with λx).
  - x=3 $\xrightarrow{\beta}$ + (+ 2 3) 4
  - =9

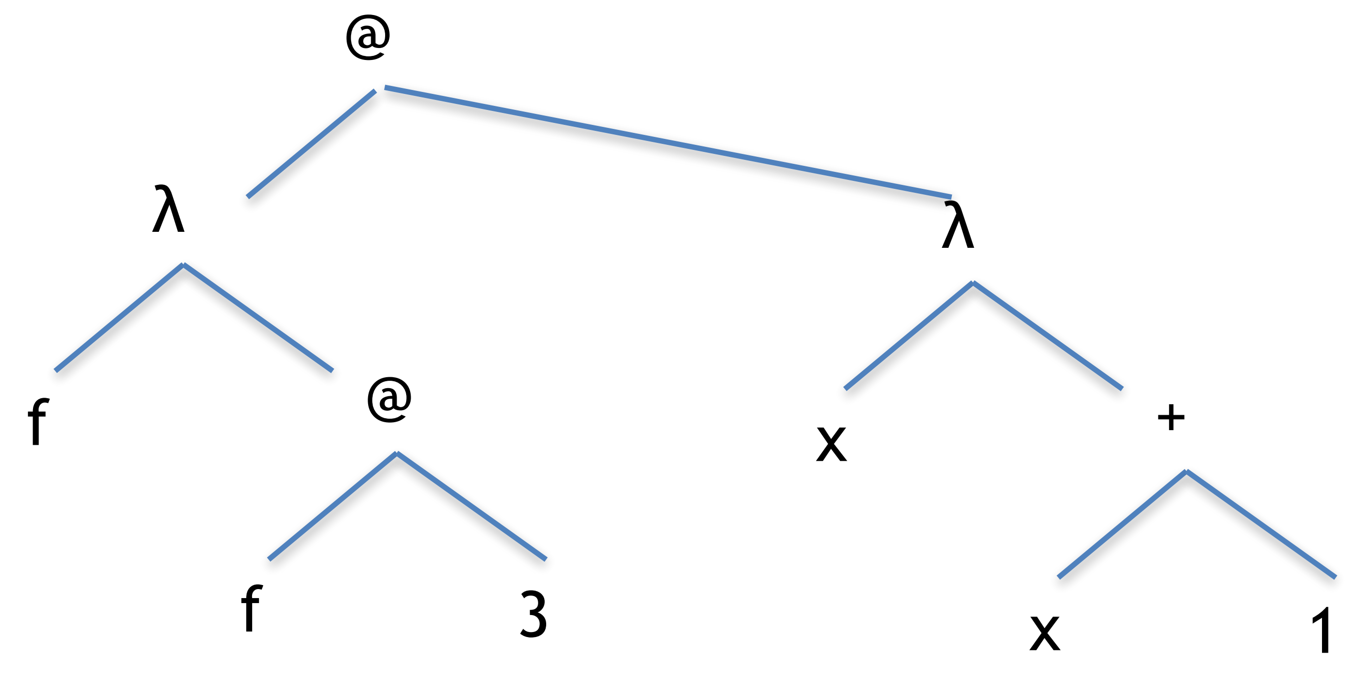
Passing lambdas as arguments

- (λf. f 3) (λx. + x 1)
- Argument is a function (lambda)
  - β reduction replaces free occurrences of f.
  - So we get:
    - (λx. + x 1) 3
  - Another β reduction follows:
    - + 3 1 = 4

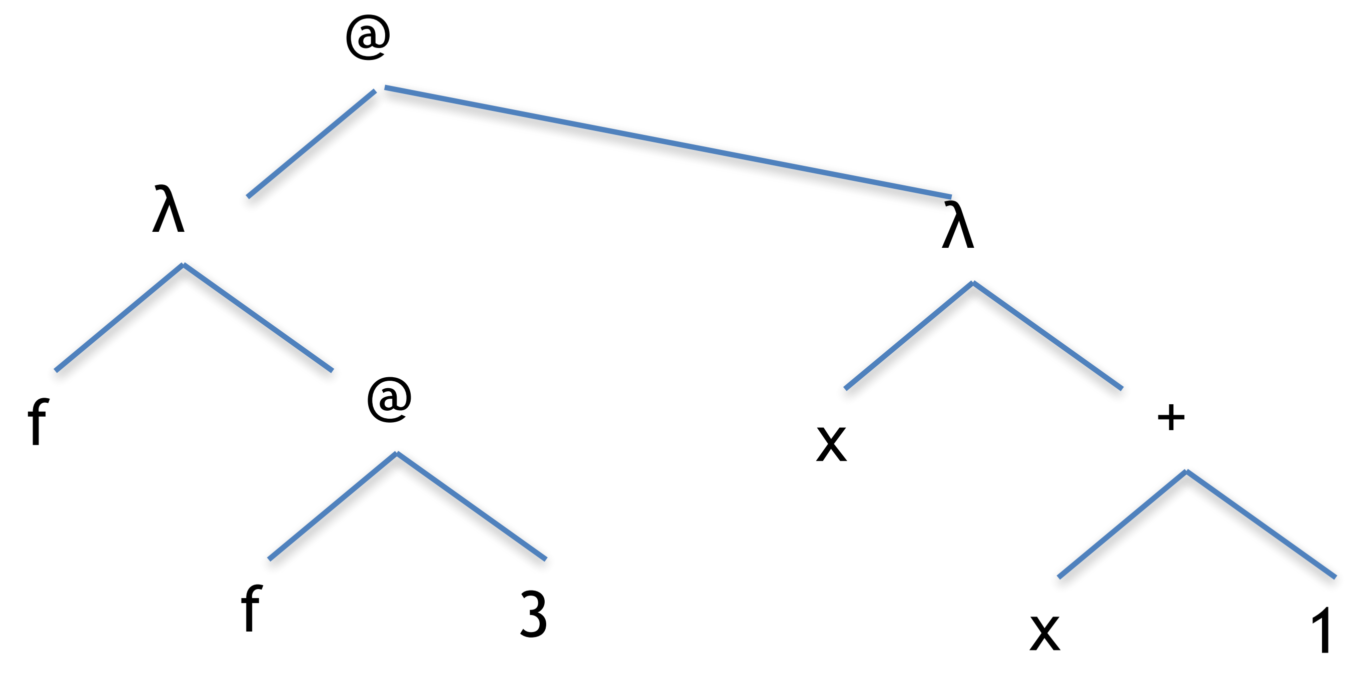
Passing Lambdas as Arguments

- Is this a strange thing to do?
  - No, it is an ENORMOUSLY powerful thing in programming
  - Usually modify functionality by passing **data**
  - Can modify functionality by passing **code**
  - GPUs are often programmed in this way
- Extremely difficult to do in imperative programming
- Simple to do in functional programming

(λf. f 3) (λx. + x 1)



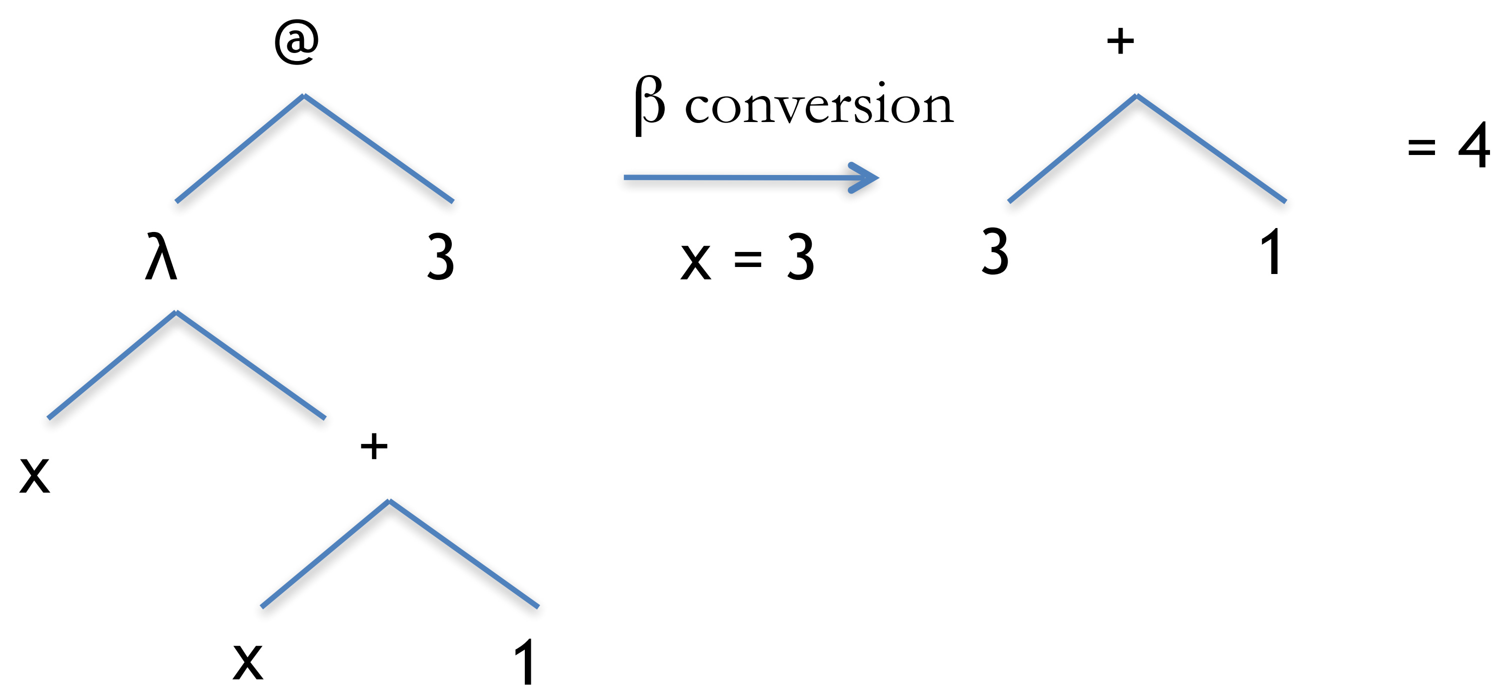
$(\lambda f.\ f\ 3)\ (\lambda x.\ +\ x\ 1)$



$(\lambda f.\ f\ 3)\ (\lambda x.\ +\ x\ 1)$

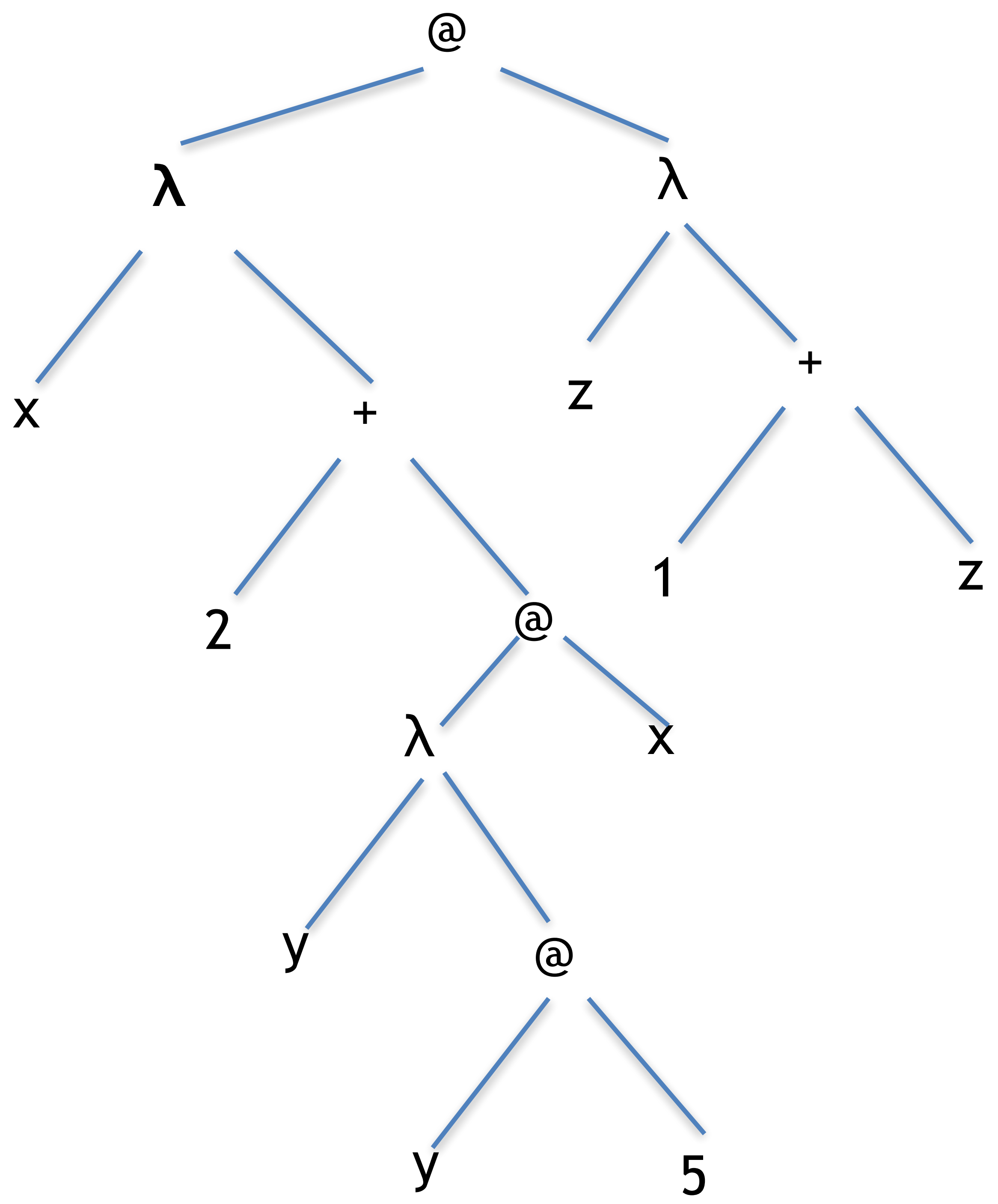
β conversion

$f = (\lambda x.\ +\ x\ 1)$



Another Example of Passing lambdas

- (λx. + 2 (λy. y 5) x)   (λz. + 1 z)

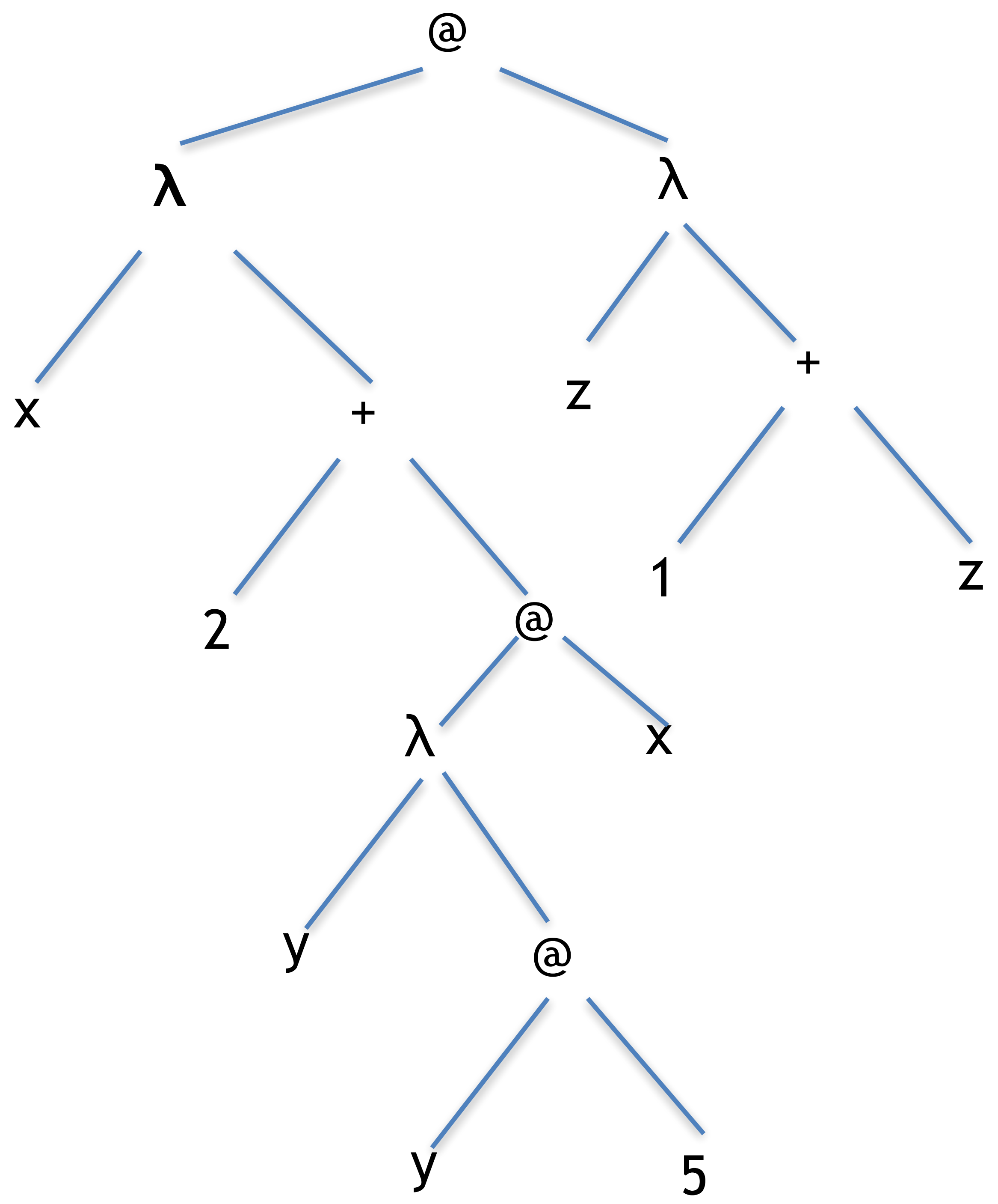


Another Example of Passing lambdas

- (λx. + 2 (λy. y 5) x)   (λz. + 1 z)

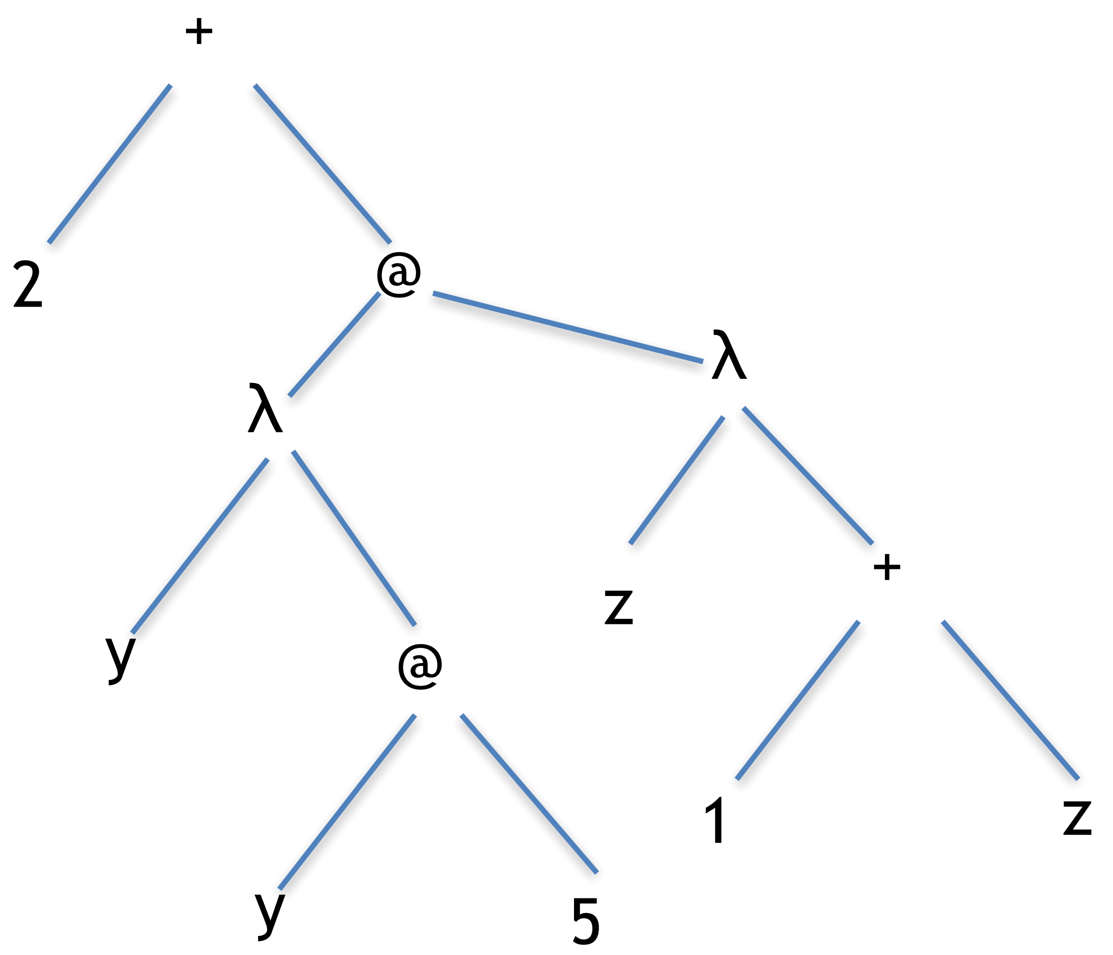
$$\xrightarrow{\beta}$$

x = λz. + 1 z



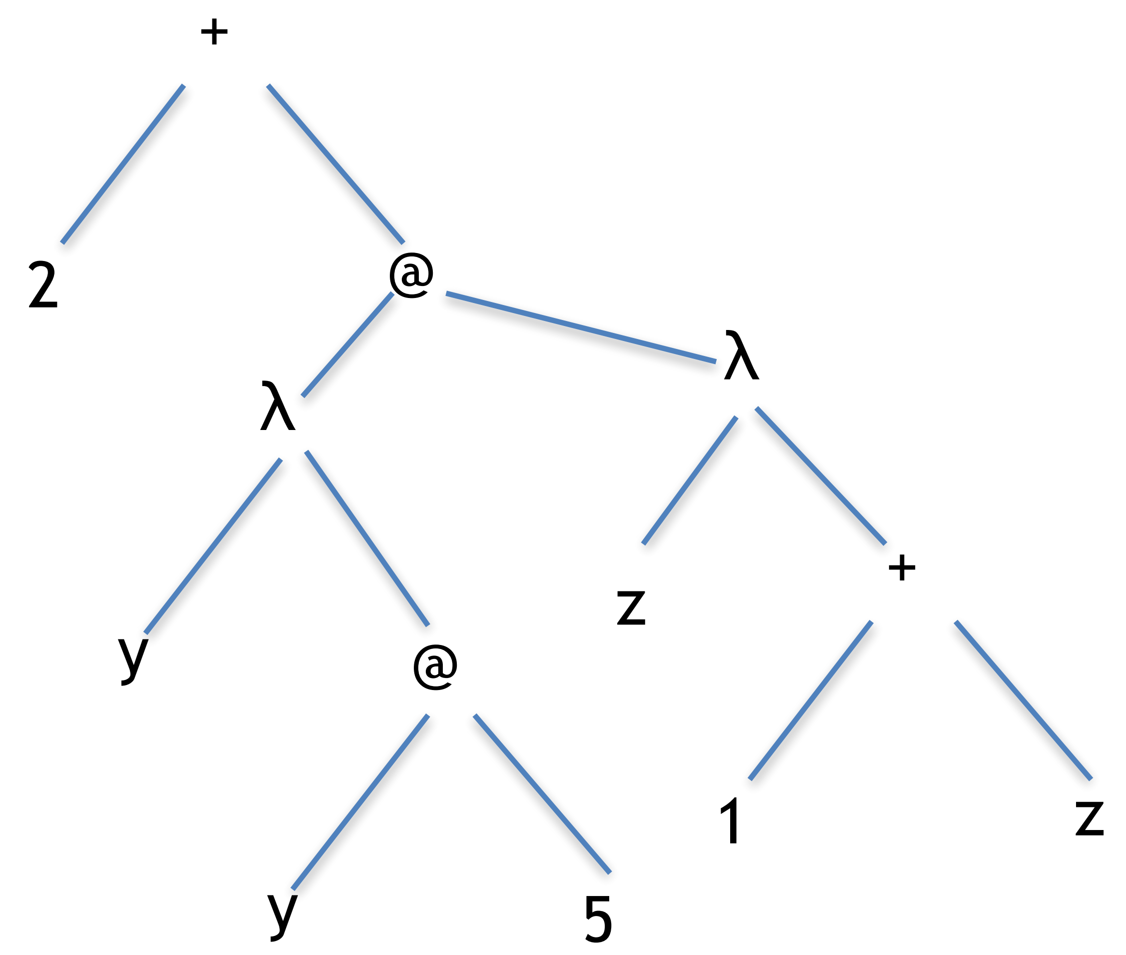
Another Example of Passing lambdas

- (+ 2 (λy. y 5) (λz. + 1 z))



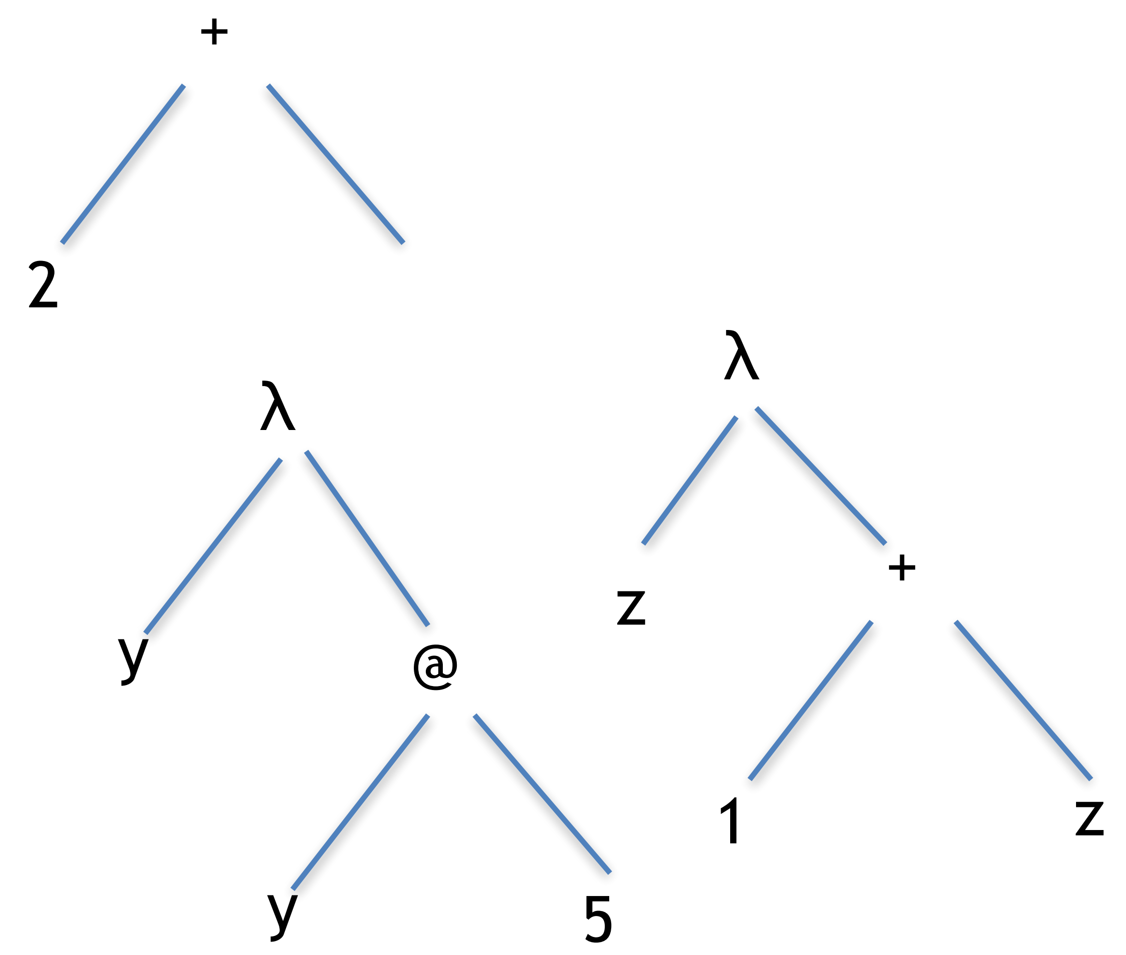
Another Example of Passing lambdas

- (+ 2 (λy. y 5) (λz. + 1 z))



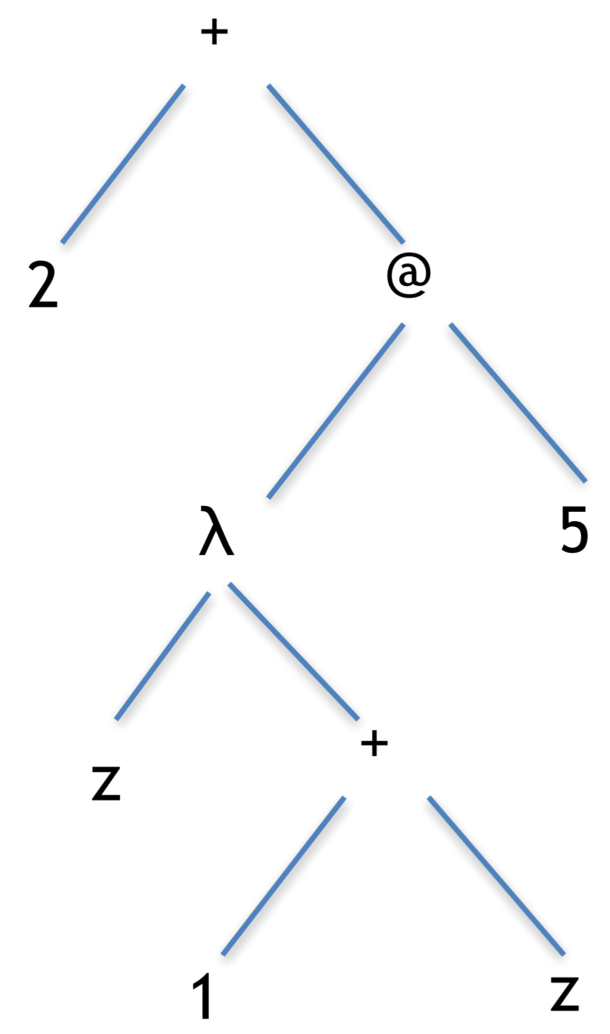
Another Example of Passing lambdas

- (λx. + 2 (λy. y 5) x)   (λz. + 1 z)



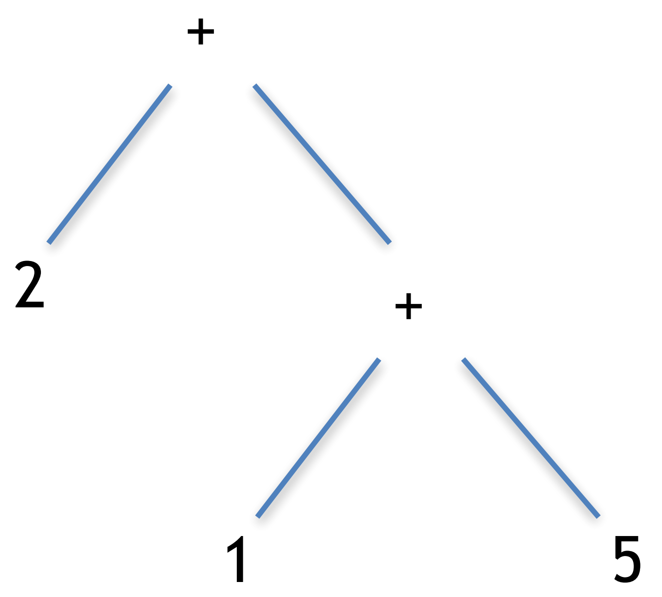
Another Example of Passing lambdas

- (+ 2 ((λz. + 1 z) 5))



Another Example of Passing lambdas

- (+ 2 (+ 1 5))



- Remember: Functions in λ calculus and ASTs (usually) don't have names
- Racket can use them
  - Useful for reusing functions
  - Useful for debugging
  - Slightly more longwinded
    - > (define t (lambda (f) (f 3)))
    - > (t   (lambda (x) (+ x 1))   )
    - 4

- > (define t2 (lambda (f) (f 2 3)))

  > (t2 +)

  (+ 2 3)

  5

  > (t2 7)

  Error: attempt to call a non-procedure [**(7 2 3)**]
- Lesson?
  – Anything can be passed as a parameter: numbers, variables, functions, operators
  – Syntax the same in lambda calculus, AST and Racket
  – Not consistent in imperative programming
    - Very different when passing a function to a function

- Formal Notation for β reduction:
  - $(\lambda x.\ E)a \xrightarrow{\beta} E[a/x]$
  - Meaning: in E, replace free occurrences of x with a
- Consider: $(\lambda x.\ +\ x\ 1)$ and $(\lambda y.\ +\ y\ 1)$
  - Are they the same?
  - Yes - names don't matter.
  - Converting one into another: **<u>α-conversion</u>**
  - E.g. $(\lambda x.\ +\ x\ 1) \overset{\alpha}{\longleftrightarrow} (\lambda y.\ +\ y\ 1)$
  - **<u>Note:</u>** bi-directional arrow: two way process

- However, if we replace x with y in:
  - (λx. + x y)
  - We get: (λy. + y y)
  - Not correct. Why?
    - Because y is free in (λx. + x y)
  - What about:
    - (λx. + x (λy. + y 1) 2) $\xleftrightarrow{\alpha}$ (λy. + y (λy. + y 1) 2)
    - This is fine
    - y is NOT free in the body of the lambda on left side.
- Formal Definition:
  - λx. E $\xleftrightarrow{\alpha}$ λy. E[y/x], **IF** y does not already exist free in **E**.

Utility of α-conversion

- (λf. (λx. f (f x))) x
- β-reduction=>(λx. x (x x))
  - Erroneous.
  - What to do?
- Use α-conversion to avoid confusion:
  - convert x into y inside the nested lambda.
  - (λf. (λy. f (f y))) x
  - β-reduction=> (λy. x (x y))
  - Correct

δ-conversion and Normal Form

- $(\lambda x.\ (+\ x\ 1))\ 2$
  - $\xrightarrow{\beta}$ $(+\ 2\ 1)$
  - $\xrightarrow{\delta}$ 3
- $(F\ a1\ a2) \xrightarrow{\delta}$ result, where F is a built in operator
- β-reduction puts values in, δ-conversion evaluates them
- The result after full evaluation is said to be in **Normal form**
  - E.g. $(+2\ 1) = 3$ is in Normal form
  - No more redexes left.

More examples of δ-conversion

- reducing redexes to normal form e.g.:
  - (* 3 (+ 5 2))
  - $\overrightarrow{\delta}$ (* 3 7)
  - $\overrightarrow{\delta}$ 21
  - 21 is in normal form

β-reduction – an interesting example

- $(\lambda f.\ (\lambda x.\ f\ \ 4\ x))\ \ (\lambda yx.\ +\ x\ y)\ \ 3$
  - $(\lambda f.\ (\lambda x.\ \ \ f\ \ \ 4\ x))\ \ (\lambda yx.\ +\ x\ y)\ \ 3$
  - $\xrightarrow{\beta}\ (\lambda x.\ (\lambda yx.\ +\ x\ y)\ 4\ x)\ \ 3$
  - $\xrightarrow{\beta}\ (\lambda yx.\ +\ x\ y)\ 4\ 3$
  - $\xrightarrow{\beta}\ (+\ 3\ 4)$
  - $\xrightarrow{\delta}\ 7$
- Racket Code
  - (define Lf (lambda (f) (lambda (x) (f 4 x))))
  - (define Lyx (lambda (y x) (+ x y)))
  - ((Lf Lyx) 3)

When to evaluate arguments- The effect

- Consider function
  - D: (λx. x x)
  - In Racket: (define D (lambda(x) (x x)))
- Evaluate D D
  - (λx. x x) (λx. x x)
  - $\xrightarrow{\beta}$ (λx. x x) (λx. x x)
  - $\xrightarrow{\beta}$ (λx. x x) (λx. x x)
  - Infinite calls
  - Try it in Racket using (D D)

When to evaluate arguments- The effect

- Consider $(\lambda x.\ 3)\ 7$
  - $\overrightarrow{\beta}\quad 3$
  - Result is 3; no matter what the argument is.
  - Evaluating the argument is needless.
- Consider $(\lambda x.\ 3)\ (D\ D)$
  - Evaluate the argument first? Infinite calls.
  - Otherwise, the answer is just 3.

Order of evaluating arguments

- How do we evaluate simple expressions?
  - So far "innermost"
  - e.g. (+ (* 2 3) 4)
- Applicative Order (Eager Evaluation):
  - "leftmost innermost".
  - i.e. try to evaluate the leftmost redex;
  - Immediately go to the innermost level of nesting
  - (λxy. + x y) (+ 1 2) (+ 3 4)
  - =(λxy. + x y) 3 (+ 3 4)
  - =(λxy. + x y) 3 7

Lazy Evaluation/Normal Order

- Back to (λx. 3) (D D):
  - Applicative Order forces evaluation of (D D) even though it is **not** needed
  - Arguments are evaluated EXACTLY ONCE
- Another Strategy: Normal Order
  - Reduce "leftmost outermost". i.e. work with the outermost bracket level whenever possible.
  - (λx. + x 1) (+ 2 3)
  - $\xrightarrow{\beta}$ (+ (+ 2 3) 1)
  - Can not work at the outermost level now. So reduce the inner (nested) redex.
  - =(+ 5 1) = 6

- + is a "strict" function:
  - Requires all its arguments before proceeding further
  - Forces evaluation of arguments even in lazy evaluation

- (λx. 3) (D D) with Normal Order
  - 3
  - (D D) not evaluated

Implications

- Applicative Order *can* cause infinite calls, and evaluate arguments needlessly
- It evaluates arguments **exactly** once
  - regardless of whether or not they are needed
- Normal Order only evaluates arguments when necessary
- It evaluates arguments **zero or more** times
  - this **might** be more inefficient
- The dream: Fully Lazy Evaluation
  - evaluate arguments **zero or one** times
  - possible, but beyond the scope of this module

Another Example

- (λx. + x x) (* 6 2)
- Normal Order β reduction:
  - + (* 6 2) (* 6 2)
  - + 12 (* 6 2)
  - + 12 12 = 24
- Applicative Order β reduction:
  - Evaluate argument *before* β reduction; we get 12
  - + 12 12
  - =24