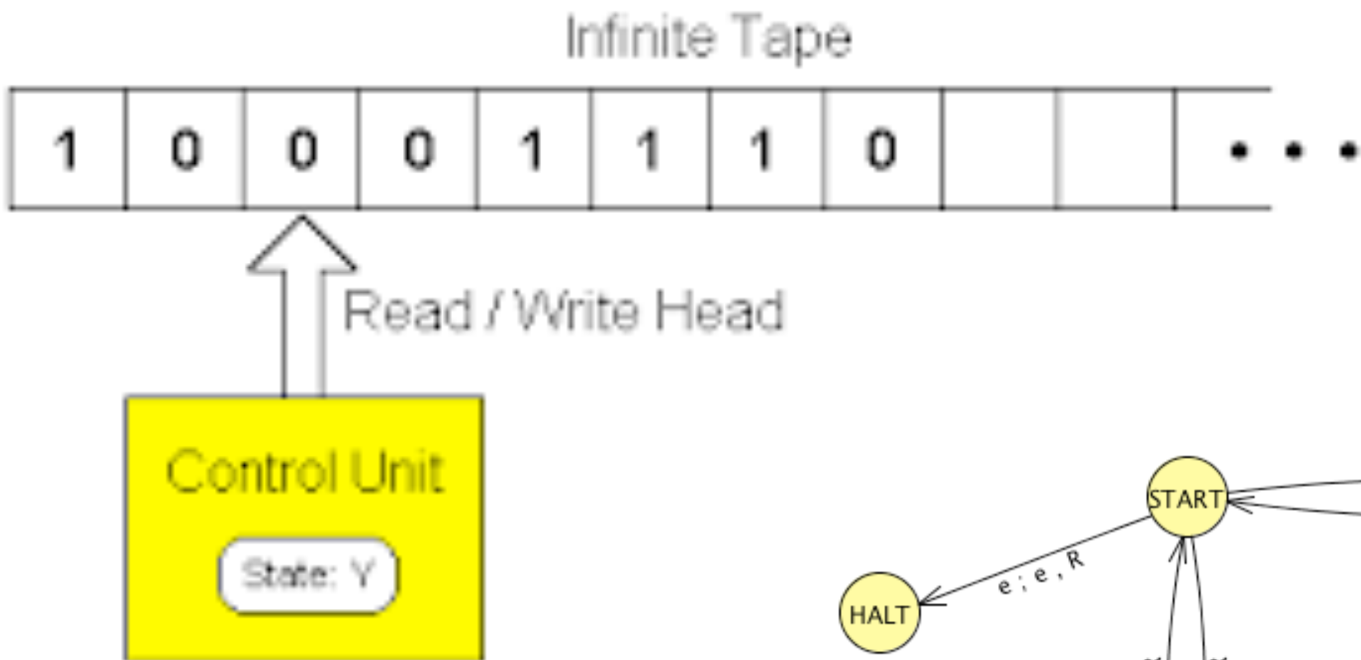
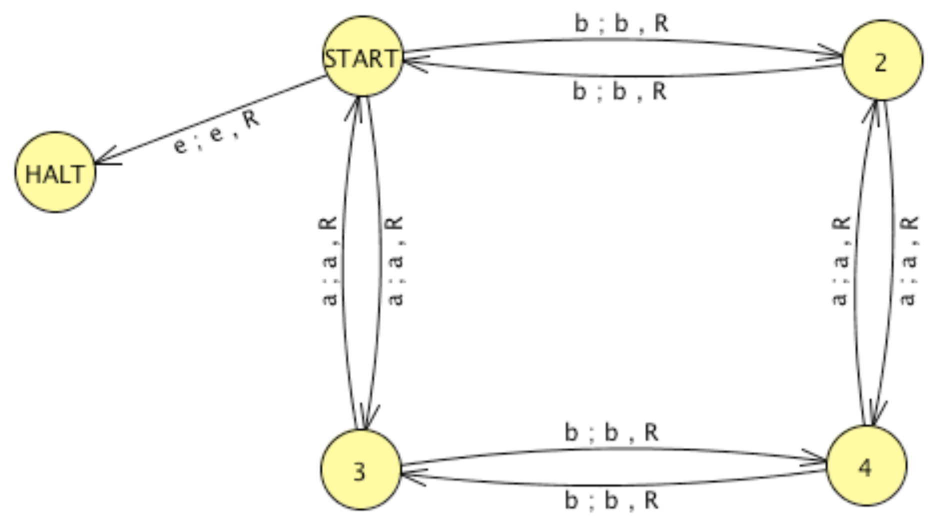
# CMSC 330: Organization of Programming Languages

## Lambda Calculus

# Turing Machine



# Turing Completeness

- Turing machines are the most powerful description of computation possible
  - They define the Turing-computable functions
- A programming language is Turing complete if
  - It can map every Turing machine to a program
  - A program can be written to emulate a Turing machine
  - It is a superset of a known Turing-complete language
- Most powerful programming language possible
  - Since Turing machine is most powerful automaton

# Programming Language Expressiveness

- So what language features are needed to express all computable functions?
  - What's a minimal language that is Turing Complete?
- Observe: some features exist just for convenience
  - Multi-argument functions foo ( a, b, c )
    - Use currying or tuples
  - Loops while (a < b) …
    - Use recursion
  - Side effects a := 1
    - Use functional programming pass "heap" as an argument to each function, return it when with function's result:
      effectful : 'a → 's → ('s * 'a)

# Programming Language Expressiveness

- It is not difficult to achieve Turing Completeness
  - Lots of things are ‘accidentally’ TC
- Some fun examples:
  - x86_64 `mov` instruction
  - Minecraft
  - Magic: The Gathering
  - Java Generics
- There’s a whole cottage industry of proving things to be TC
- But: What is a “core” language that is TC?

# Lambda Calculus (λ-calculus)

- Proposed in 1930s by
  - Alonzo Church
    (born in Washingon DC!)

- Formal system
  - Designed to investigate functions & recursion
  - For exploration of foundations of mathematics
- Now used as
  - Tool for investigating computability
  - Basis of functional programming languages
    - Lisp, Scheme, ML, OCaml, Haskell…

# Why Study Lambda Calculus?

- It is a "core" language
  - Very small but still Turing complete
- But with it can explore general ideas
  - Language features, semantics, proof systems, algorithms, …
- Plus, higher-order, anonymous functions (aka *lambdas*) are now very popular!
  - C++ (C++11), PHP (PHP 5.3.0), C# (C# v2.0), Delphi (since 2009), Objective C, Java 8, Swift, Python, Ruby (Procs), … (and functional languages like OCaml, Haskell, F#, …)
  - Excel, as of 2021!

# Lambda Calculus Syntax

- A lambda calculus expression is defined as

| | |
|---|---|
| e ::= x | **variable** |
| \| λx.e | **abstraction** (fun def) |
| \| e e | **application** (fun call) |

  - This grammar describes ASTs; not for parsing - ambiguous!
  - Lambda expressions also known as lambda **terms**

- λx.e is like `(fun x -> e)` in OCaml

That's it! Nothing but higher-order functions

# Lambda Calculus Syntax Ambiguity

- How is parsing ambiguous?
- Let's try: λx.x x

E → V | L | A

L → λV.E

A → E E

V → v | ε

L

λ V . A

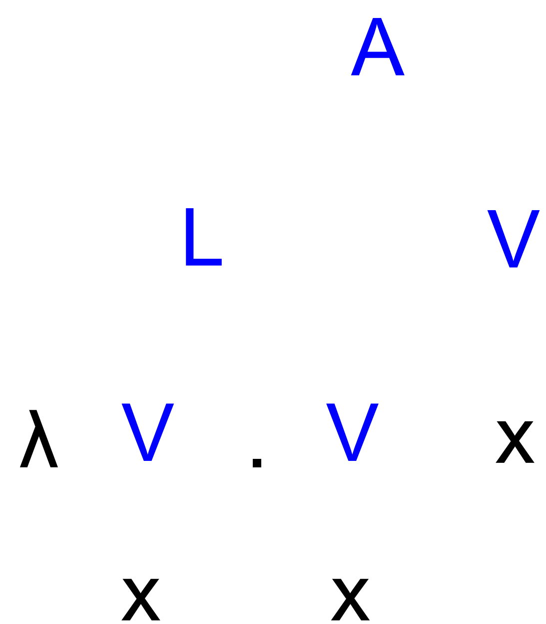
V V V

x x x

# Lambda Calculus Syntax Ambiguity

- How is parsing ambiguous?
- Let’s try: λx.x x

E → V | L | A
L → λV.E
A → E E
V → v | ε



# Lambda Calculus Syntax

- While this means that our grammar is not so useful for *parsing*, it is still useful for write LC terms if we follow some conventions
- Almost all literature you will find uses two syntactic conventions
- We add a third convention that is very common 'syntactic sugar' for ease of reading larger LC terms

# Disambiguating: Three Conventions

- Scope of λ extends as far right as possible
  - Subject to scope delimited by parentheses
  - λx. λy.x y is same as λx.(λy.(x y))
- Function application is left-associative
  - x y z is (x y) z
  - Same rule as OCaml
- As a convenience, we use the following "syntactic sugar" for local declarations
  - let x = e1 in e2 is short for (λx.e2) e1

# Warmup Quiz

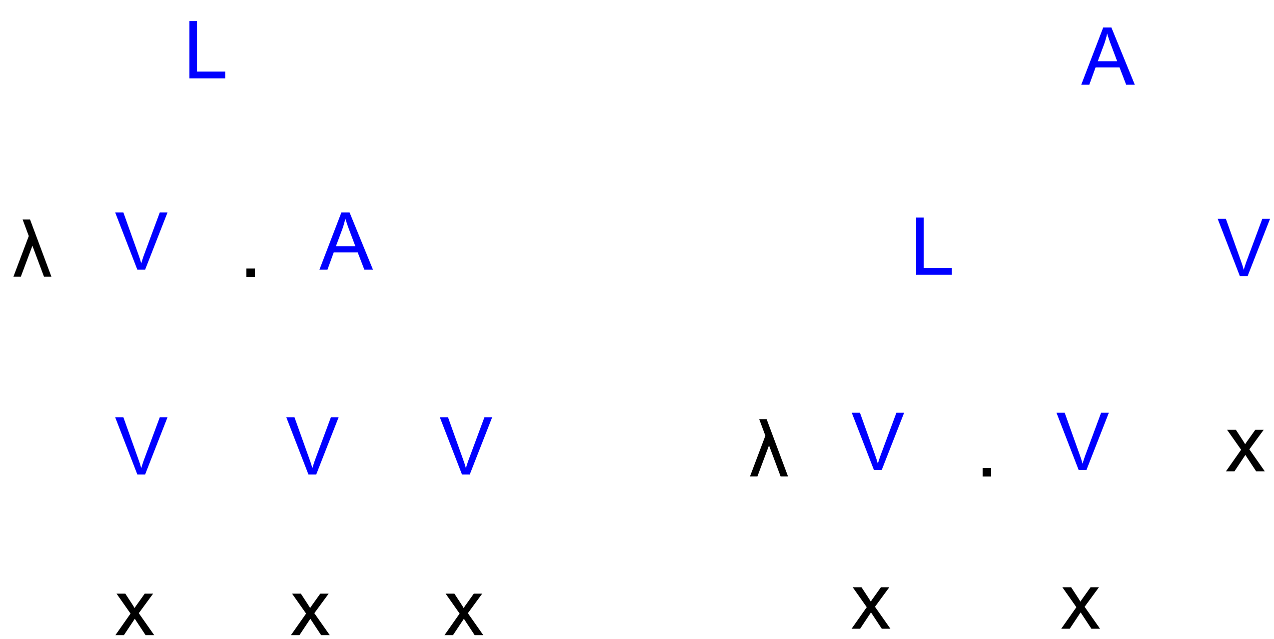
- Revisiting λx.x x considering our conventions
- Which parse tree is it?

E → V | L | A

L → λV.E

A → E E

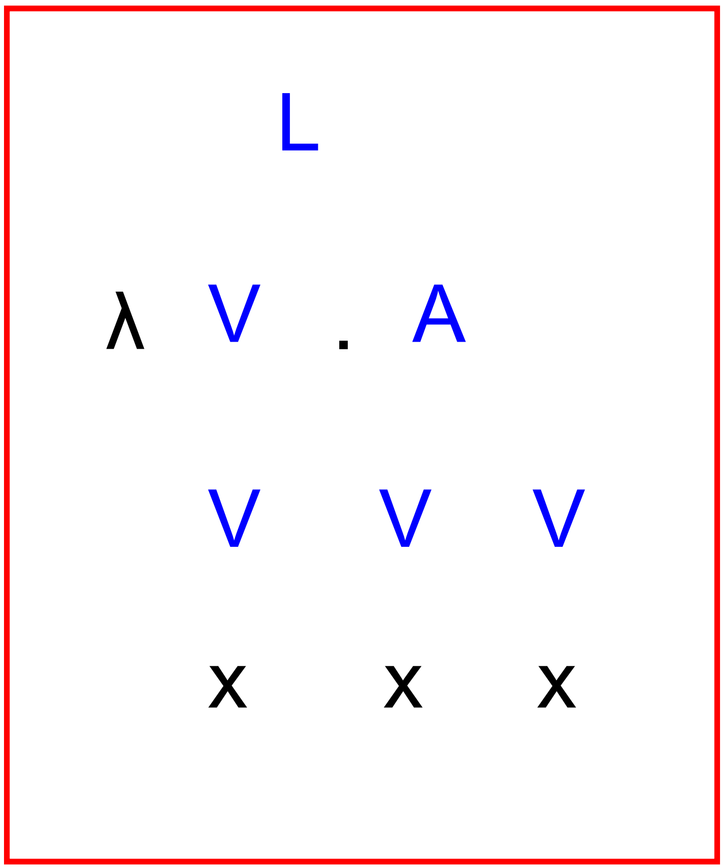
V → v | ε

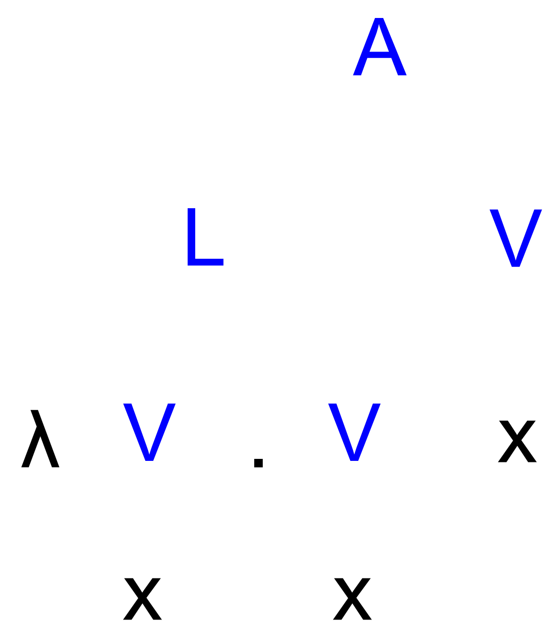


# Warmup Quiz

- Revisiting λx.x x considering our conventions
- Which parse tree is it?

E → V | L | A
L → λV.E
A → E E
V → v | ε



# Quiz #1

***λx. (y z)*** and ***λx.y z*** are equivalent

A. True
B. False

# Quiz #1

***λx. (y z)*** and ***λx.y z*** are equivalent

A. **True**
B. False

# Quiz #2

This term is equivalent to which of the following?

$$\lambda x.x\ a\ b$$

A. $(\lambda x.x)\ (a\ b)$
B. $(((\lambda x.x)\ a)\ b)$
C. $\lambda x.(x\ (a\ b))$
D. $(\lambda x.((x\ a)\ b))$

# Quiz #2

This term is equivalent to which of the following?

$$\lambda x.x\ a\ b$$

**A.** $(\lambda x.x)\ (a\ b)$
**B.** $(((\lambda x.x)\ a)\ b)$
**C.** $\lambda x.(x\ (a\ b))$
**D.** $(\lambda x.((x\ a)\ b))$

# But what does it mean?

- Many ways to define the semantics of LC
- We will look at two
  - Operational Semantics
  - Definitional Interpreter

# Lambda Calculus Semantics

- Evaluation: All that's involved are function calls (λx.e1) e2
  - Evaluate e1 with x replaced by e2
- This application is called beta-reduction
  - (λx.e1) e2 → e1[x:=e2]
    - e1[x:=e2] is e1 with occurrences of x replaced by e2
    - This operation is called *substitution*
      - **Replace** formals with actuals
      - Instead of using environment to map formals to actuals
  - We allow reductions to occur *anywhere* in a term
    - Order reductions are applied does not affect final value!
- When a term cannot be reduced further it is in beta normal form

# Beta Reduction Example

- (λx.λz.x z) y
  - → (λx.(λz.(x z))) y    // since λ extends to right
  - → (λx.(λz.(x z))) y    // apply (λx.e1) e2 → e1[x:=e2]
    // where e1 = λz.(x z), e2 = y
  - → λz.(y z)    // final result

Parameters
- Formal
- Actual

- Equivalent OCaml code
  - (fun x -> (fun z -> (x z))) y → fun z -> (y z)

# Big-Step Operational Semantics

- Beta reduction says how to evaluate a single call
  - It doesn't say how to evaluate a term with many function calls in it
- We can use operational semantics to "fully evaluate" a term in one "big step"

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

Beta reduction, here

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e2 \Downarrow e4 \qquad e3[x:=e4] \Downarrow e5}{e1\ e2 \Downarrow e5}$$

# Two Varieties

- There are two common variants of big-step semantics
  - *Eager* evaluation (aka *strict*, or *call by value*)
  - *Lazy* evaluation (aka *call by name*)

# Eager

- Notice that we evaluated the argument e2 before performing the beta-reduction
  - This is the first version we saw
- Hence, *eager*

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e2 \Downarrow e4 \qquad e3[x:=e4] \Downarrow e5}{e1\ e2 \Downarrow e5}$$

# Lazy

- Alternatively, we could have performed beta reduction *without* evaluating e2; use it as is
  - Hence, *lazy*

$$\frac{}{(\lambda x.e1) \Downarrow (\lambda x.e1)}$$

$$\frac{e1 \Downarrow (\lambda x.e3) \qquad e3[x:=e2] \Downarrow e4}{e1\ e2 \Downarrow e4}$$

# Small Step Semantics

- Operational semantics rules we have seen have always been ”big step”, i.e., complete evaluation
  - e ⇓ e’ says that e will *terminate* as e’
- This is a little unsatisfying
  - It doesn’t account for nontermination
  - It doesn’t identify where a program fails to progress
- Small-step semantics addresses these problems
  - e → e’ in small-step says e **takes one step** to e’
  - We say a term e1 can be *beta-reduced* to term e2 if e1 steps to e2 after one or more steps

# Small-Step Rules of LC

- Here are the "small-step" (→) rules:

$$\frac{e1 \rightarrow \mathbf{e2}}{(\lambda x.e1) \rightarrow (\lambda x.\mathbf{e2})}$$

$$\frac{e2 \rightarrow \mathbf{e3}}{e1\ e2 \rightarrow e1\ \mathbf{e3}}$$

$$\frac{e1 \rightarrow \mathbf{e3}}{e1\ e2 \rightarrow \mathbf{e3}\ \ e2}$$

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[x:=e2]}$$

# Evaluation Strategies

- These rules are highly flexible
  - It might be that for a given program, there are several possible rules that could apply
- Typically, a programming language will choose an *evaluation strategy* which is described by using only a subset of these rules. Examples:
  - Call by Value
  - Call by Need
  - Partial Evaluation

# Call by Value

- Before doing a beta reduction, we make sure the argument cannot, itself, be further evaluated
- This is known as call-by-value (CBV)
  - This is the Eager big step approach

$$\frac{e1 \rightarrow \mathbf{\mathit{e3}}}{e1\ e2 \rightarrow \mathbf{\mathit{e3}}\ \ e2}$$

$$\frac{e2 \rightarrow \mathbf{\mathit{e3}}}{e1\ e2 \rightarrow\ e1\ \mathbf{\mathit{e3}}}$$

$$\frac{e = (\lambda x.e2) \text{ or } e = y}{(\lambda x.e1)\ e \rightarrow e1[x{:=}e]}$$

# Beta Reductions (CBV)

- $(\lambda x.x)\ z \rightarrow z$
- $(\lambda x.y)\ z \rightarrow y$
- $(\lambda x.x\ y)\ z \rightarrow z\ y$
  - A function that applies its argument to y

# Beta Reductions (CBV)

- (λx.x y) (λz.z) → (λz.z) y → y
- (λx.λy.x y) z → λy.z y
  - A curried function of two arguments
  - Applies its first argument to its second
- (λx.λy.x y) (λz.zz) x → (λy.(λz.zz)y)x → (λz.zz)x → x x

# Quiz #3

**(λx.y) z** can be beta-reduced to

A. **y**
B. **y z**
C. **z**
D. cannot be reduced

# Quiz #3

$(\lambda x.y)\ z$ can be beta-reduced to

A. $y$
B. $y\ z$
C. $z$
D. cannot be reduced

# Quiz #4

Which of the following reduces to λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
c) (λy. y) (λx. λz. z) w
d) (λy. λx. z) z (λz. z)

# Quiz #4

Which of the following reduces to λz. z?

a) (λy. λz. x) z
b) (λz. λx. z) y
**c) (λy. y) (λx. λz. z) w**
d) (λy. λx. z) z (λz. z)

# Evaluation Order

- The CBV rules we saw permit small-stepping either the function part or the argument part
  - If both are possible, the rules allow either one

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2} \qquad \frac{e2 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow e1\ \boldsymbol{e3}}$$

- Here's how we would require left-to-right order

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2} \qquad \frac{e1 = y \quad \text{or} \quad e1 = \lambda x.e \qquad e2 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow e1\ \boldsymbol{e3}}$$

  - The second rule prohibits evaluating e2 except when e1 cannot be evaluated further

# Call by Name

- Instead of the CBV strategy, we can specifically choose to perform beta-reduction *before* we evaluate the argument
- This is known as call-by-name (CBN)
  - This is the Lazy small-step approach

$$\frac{e1 \rightarrow \boldsymbol{e3}}{e1\ e2 \rightarrow \boldsymbol{e3}\ e2}$$

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[x:=e2]}$$

# CBN Reduction

- CBV
  - (λz.z) ((λy.y) x) → (λz.z) x → x
- CBN
  - (λz.z) ((λy.y) x) → (λy.y) x → x

# Beta Reductions (CBN)

(λx.x (λy.y)) (u r) →

(λx.(λw. x w)) (y z) →

# Beta Reductions (CBN)

(λx.x (λy.y)) (u r) → (u r) (λy.y)

(λx.(λw. x w)) (y z) → (λw. (y z) w)

# Why Does This Matter?

- The rules we just showed are very common for programming languages based on LC
  - CBV is the most common (e.g. OCaml, Java)
  - CBN does come up (Haskell uses a variant known as "call-by-need") but is much less common
- Interestingly: more programs terminated under call-by-name. Can you think of why?
  - Consider: (λx.e2) e1,
  - What if e1 would never terminate, but e2 would?

# Evaluating Within a Function

- Our original rules had evaluation *under* the lambda
- Where does this help us?

$$\frac{e1 \rightarrow \mathbf{e2}}{(\lambda x.e1) \rightarrow (\lambda x.\mathbf{e2})}$$

$$\frac{e2 \rightarrow \mathbf{e3}}{e1\ e2 \rightarrow e1\ \mathbf{e3}} \qquad \frac{e1 \rightarrow \mathbf{e3}}{e1\ e2 \rightarrow \mathbf{e3}\ \ e2}$$

$$\frac{}{(\lambda x.e1)\ e2 \rightarrow e1[x:=e2]}$$

# Partial Evaluation

- That rule is useful when you have a beta-reduction *under* a lambda:
  - (λy.(λz.z) y x) → (λy.y x)
- Called partial evaluation
  - Can combine with CBN or CBV (just add in the rule)
  - In practical languages, this evaluation strategy is employed in a limited way, as compiler optimization

```
int foo(int x) {            int foo(int x) {
  return 0+x;        →        return x;
}                           }
```

# Static Scoping & Alpha Conversion

- Lambda calculus uses static scoping
- Consider the following
  - (λx.x (λx.x)) z → ?
    - The rightmost "x" refers to the second binding
  - This is a function that
    - Takes its argument and applies it to the identity function
- This function is "the same" as (λx.x (λy.y))
  - Renaming bound variables consistently preserves meaning
    - This is called alpha-renaming or alpha conversion
  - Ex. λx.x = λy.y = λz.z     λy.λx.y = λz.λx.z

# Quiz #5

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) $\lambda y.\ y\ y$

b) $\lambda z.\ y\ z$

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

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

# Quiz #5

Which of the following expressions is alpha equivalent to (alpha-converts from)

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

a) λy. y y

b) λz. y z

**c) (λx. λz. x z) y**

d) (λx. λy. x y) z

# Getting Serious about Substitution

- We have been thinking informally about substitution, but the details matter
- So, let's carefully formalize it, to help us see where it can get tricky!

# Defining Substitution

- Use recursion on structure of terms
  - x[x:=e] = e // Replace x by e
  - y[x:=e] = y // y is different than x, so no effect
  - (e1 e2)[x:=e] = (e1[x:=e]) (e2[x:=e])
    // Substitute both parts of application
  - (λx.e')[x:=e] = λx.e'
    - In λx.e', the x is a parameter, and thus a local variable that is different from other x's. Implements static scoping.
    - So the substitution has no effect in this case, since the x being substituted for is different from the parameter x that is in e'
  - (λy.e')[x:=e] = ?
    - The parameter y does not share the same name as x, the variable being substituted for
    - Is λy.(e'[x:=e]) correct? No…

# Variable Capture

- How about the following?
  - (λx.λy.x y) y → ?
  - When we replace y inside, we don't want it to be captured by the inner binding of y, as this violates static scoping
  - I.e., (λx.λy.x y) y ≠ λy.y y
- Solution
  - (λx.λy.x y) is "the same" as (λx.λz.x z)
    - Due to alpha conversion
  - So alpha-convert (λx.λy.x y) y to (λx.λz.x z) y first
    - Now (λx.λz.x z) y → λz.y z

# Completing the Definition of Substitution

- Recall: we need to define (λy.e')[x:=e]
  - We want to avoid capturing (free) occurrences of y in e
  - Solution: alpha-conversion!
    - Change y to a variable w that does not appear in e' or e

      (Such a w is called fresh)
    - Replace all occurrences of y in e' by w.
    - Then replace all occurrences of x in e' by e!
- Formally:

  (λy.e')[x:=e] = λw.((e' [y:=w]) [x:=e]) (w is fresh)

# Beta-Reduction, Again

- Whenever we do a step of beta reduction
  - (λx.e1) e2 → e1[x:=e2]
  - We must alpha-convert variables as necessary
  - Sometimes performed implicitly (w/o showing conversion)

- Examples
  - (λx.λy.x y) y = (λx.λz.x z) y → λz.y z   // y → z
  - (λx.x (λx.x)) z = (λy.y (λx.x)) z → z (λx.x)   // x → y

# Quiz #6

Beta-reducing the following term produces what result?

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

A. y (λz.z y)
B. z (λy.y z)
C. y (λy.y y)
D. y y

# Quiz #6

Beta-reducing the following term produces what result?

(λx.x λy.y x) y

A. **y (λz.z y)**
B. z (λy.y z)
C. y (λy.y y)
D. y y