# Lambda Calculus
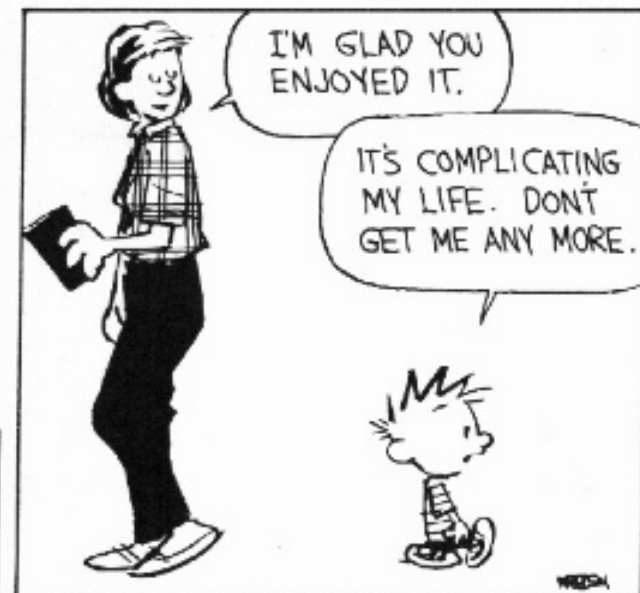
λ

# Plan



- Introduce lambda calculus
  - Syntax
  - Substitution
  - Operational Semantics (… with contexts!)
  - Evaluations strategies
  - Equality
- Later:
  - Relationship to programming languages
  - Study of types and type systems

# Lambda Background

- Developed in 1930's by <span style="color:magenta">Alonzo Church</span>
- Subsequently studied by many people
  - Still studied today!
- Considered the "testbed" for procedural and functional languages
  - Simple
  - Powerful
  - Easy to extend with new features of interest
  - Lambda:PL :: Turning Machine:Complexity
  - Somewhat like a crowbar …

"Whatever the next 700 languages turn out to be, they will surely be variants of lambda calculus."

(Landin '66)

# Lambda Syntax

- The $\lambda$-calculus has 3 kinds of expressions (terms)

$$e ::= x \qquad \text{Variables}$$
$$| \quad \lambda x.\, e \qquad \text{Functions (}\underline{\text{abstractions}}\text{)}$$
$$| \quad e_1\, e_2 \qquad \text{Application}$$

- $\lambda x.\, e$ is a one-argument <u>anonymous function</u> with body $e$

- $e_1\, e_2$ is a function application

- Application associates to the left

$$x\, y\, z === (x\, y)\, z$$

- Abstraction extends far to the right

$$\lambda x.\, x\, \lambda y.\, x\, y\, z === \lambda x.\, (x\, [\lambda y.\, \{(x\, y)\, z\}])$$

# Why Should I Care?

- A language with 3 expressions? Woof!
- Li and Zdancewic. *Downgrading policies and relaxed noninterference*. POPL '05
  - Just one example of a recent PL/security paper

## 4. LOCAL DOWNGRADING POLICIES

### 4.1 Label Definition

**Definition 4.1.1 (The policy language).** *In Figure 1.*

| | | |
|---|---|---|
| Types | $\tau ::=$ | $\text{int} \mid \tau \to \tau$ |
| Constants | $c ::=$ | $c_i$ |
| Operators | $\oplus ::=$ | $+, -, =, \ldots$ |
| Terms | $m ::=$ | $\lambda x{:}\tau.\ m \mid m\ m \mid x \mid c \mid m \oplus m$ |
| Policies | $n ::=$ | $\lambda x{:}\text{int}.\ m$ |
| Labels | $l ::=$ | $\{n_1, \ldots, n_k\}\quad (k \geq 1)$ |

**Figure 1:** $\mathbb{L}_{\text{local}}$ Label Syntax

The core of the policy language is a variant of the simply-typed $\lambda$-calculus with a base type, binary operators and constants. A **downgrading policy** is a $\lambda$-term that specifies how an integer can be downgraded: when this $\lambda$-term is applied to the annotated integer, the result becomes public. A

$$\frac{\Gamma \vdash m : \tau}{\Gamma \vdash m \equiv m : \tau} \quad \text{Q-Refl}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau}{\Gamma \vdash m_2 \equiv m_1 : \tau} \quad \text{Q-Symm}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau \qquad \Gamma \vdash m_2 \equiv m_3 : \tau}{\Gamma \vdash m_1 \equiv m_3 : \tau} \quad \text{Q-Trans}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash m_1 \equiv m_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\ m_1 \equiv \lambda x{:}\tau_1.\ m_2 : \tau_1 \to \tau_2} \quad \text{Q-Abs}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \tau_1 \to \tau_2 \qquad \Gamma \vdash m_3 \equiv m_4 : \tau_1}{\Gamma \vdash m_1\ m_3 \equiv m_2\ m_4 : \tau_2} \quad \text{Q-App}$$

$$\frac{\Gamma \vdash m_1 \equiv m_2 : \text{int} \qquad \Gamma \vdash m_3 \equiv m_4 : \text{int}}{\Gamma \vdash m_1 \oplus m_3 \equiv m_2 \oplus m_4 : \text{int}} \quad \text{Q-BinOp}$$

# Lambda Celebrity Representative

- Milton Friedman?
- Morgan Freeman?
- C. S. Friedman?

# Gordon Freeman

- Best-selling PC FPS to date ...

# Examples of Lambda Expressions
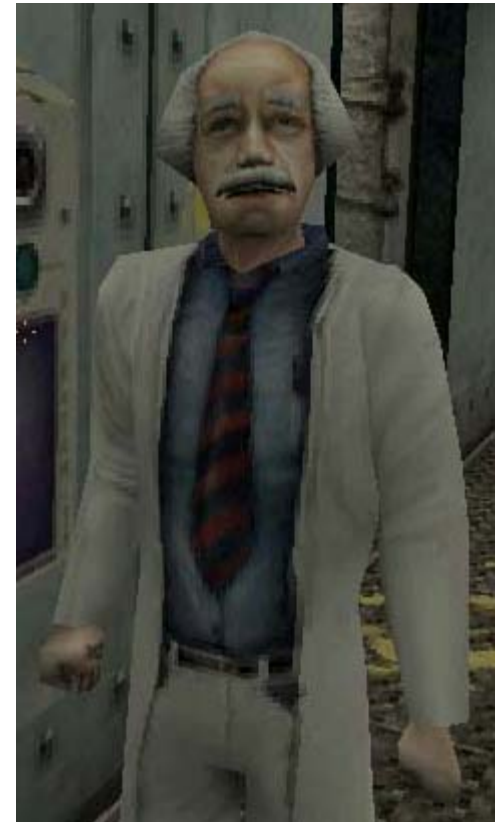
- The identity function:

$$I =_{def} \lambda x.\ x$$

- A function that, given an argument y, discards it and yields the identity function:

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

- A function that, given an function f, invokes it on the identity function:

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



*"There goes our grant money."*

# Scope of Variables

- As in all languages with variables, it is important to discuss the notion of scope
  - The scope of an identifier is the portion of a program where the identifier is accessible

- An abstraction $\lambda$x. E binds variable x in E
  - x is the newly introduced variable
  - E is the scope of x                    (unless x is shadowed)
  - We say x is bound in $\lambda$x. E
  - Just like formal function arguments are bound in the function body

# Free and Bound Variables

- A variable is said to be <u>free</u> in E if it has occurrences that are not bound in E
- We can define the free variables of an expression E recursively as follows:
  - Free(x) = {x}
  - Free($E_1$ $E_2$) = Free($E_1$) $\cup$ Free($E_2$)
  - Free($\lambda$x. E) = Free(E) – {x}
- Example: Free($\lambda$x. x ($\lambda$y. x y z)) = {z}
- Free variables are (implicitly or explicitly) declared outside the expression

# Free Your Mind!

- Just as in any language with statically-nested scoping we have to worry about variable [shadowing](shadowing)
  - An occurrence of a variable might refer to different things in different contexts
- Example in IMP with locals:

  let x = 5 in x + (let x = 9 in x) + x

- In $\lambda$-calculus:

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

# Renaming Bound Variables

- $\lambda$-terms that can be obtained from one another by renaming bound variables are considered *identical*

- This is called $\alpha$-equivalence

- Renaming bound vars is called $\alpha$-renaming

- Ex: $\lambda x.\ x$ is identical to $\lambda y.\ y$ and to $\lambda z.\ z$

- Intuition:
  - By changing the name of a formal argument and all of its occurrences in the function body, the behavior of the function *does not change*
  - In $\lambda$-calculus such functions are considered identical

# Make It Easy On Yourself

- Convention: we will always try to rename bound variables so that they are all unique
  - e.g., write $\lambda x.\ x\ (\lambda y.y)\ x$ instead of $\lambda x.\ x\ (\lambda x.x)\ x$
- This makes it easy to see the scope of bindings and also prevents confusion!

# Substitution

- The substitution of F for x in E (written [F/x]E)
  - Step 1. Rename bound variables in E and F so they are unique
  - Step 2. Perform the textual substitution of f for X in E
- Called capture-avoiding substitution
- Example: [y (λx. x) / x] λy. (λx. x) y x
  - After renaming: [y (λx. x) / x] λz. (λu. u) z x
  - After substitution: λz. (λu. u) z (y (λx. x))
- If we are not careful with scopes we might get:

  λy. (λx. x) y (y (λx. x))   ← wrong!

# The De Bruijn Notation

- An alternative syntax that avoids naming of bound variables (and the subsequent confusions)
- The De Bruijn index of a variable *occurrence* is that number of lambda that separate the occurrence from its binding lambda in the abstract syntax tree
- The De Bruijn notation replaces names of occurrences with their De Bruijn indices
- Examples:
  - $\lambda$ x. x        $\lambda$. 0
  - $\lambda$ x. $\lambda$ x. x      $\lambda$. $\lambda$. 0
  - $\lambda$ x. $\lambda$ y. y      $\lambda$. $\lambda$. 0
  - ($\lambda$ x. x x) ($\lambda$ z. z z)    ($\lambda$. 0 0) ($\lambda$. 0 0)
  - $\lambda$ x. ($\lambda$ x. $\lambda$ y. x) x    $\lambda$. ($\lambda$. $\lambda$. 1) 0

Identical terms have identical representations!

# Combinators

- A $\lambda$-term without free variables is [closed] or a [combinator]
- Some interesting combinators:

  $$I \quad = \lambda\, x.\ x$$
  $$K \quad = \lambda\, x.\ \lambda\, y.\ x$$
  $$S \quad = \lambda\, f.\ \lambda\, g.\ \lambda\, x.\ f\ x\ (g\ x)$$
  $$D \quad = \lambda\, x.\ x\ x$$
  $$Y \quad = \lambda\, f.\ (\lambda\, x.\ f\ (x\ x))\ (\lambda\, x.\ f\ (x\ x))$$

- Theorem: any closed term is equivalent to one written with just S, K and I
  - Example: $D =_\beta S\ I\ I$
  - (we'll discuss this form of equivalence later)

# Q: Music (241 / 842)

- Name the singer and his crossover 1982 album that holds (as of 2005) the record of being the best-selling album of all-original material in the US (26 times platinum, 37 weeks as Billboard #1). Much of that success was the result of the singer's use of the MTV music video.

- Name two of the three rules given for the pet Gizmo in the 1984 movie **Gremlins**.
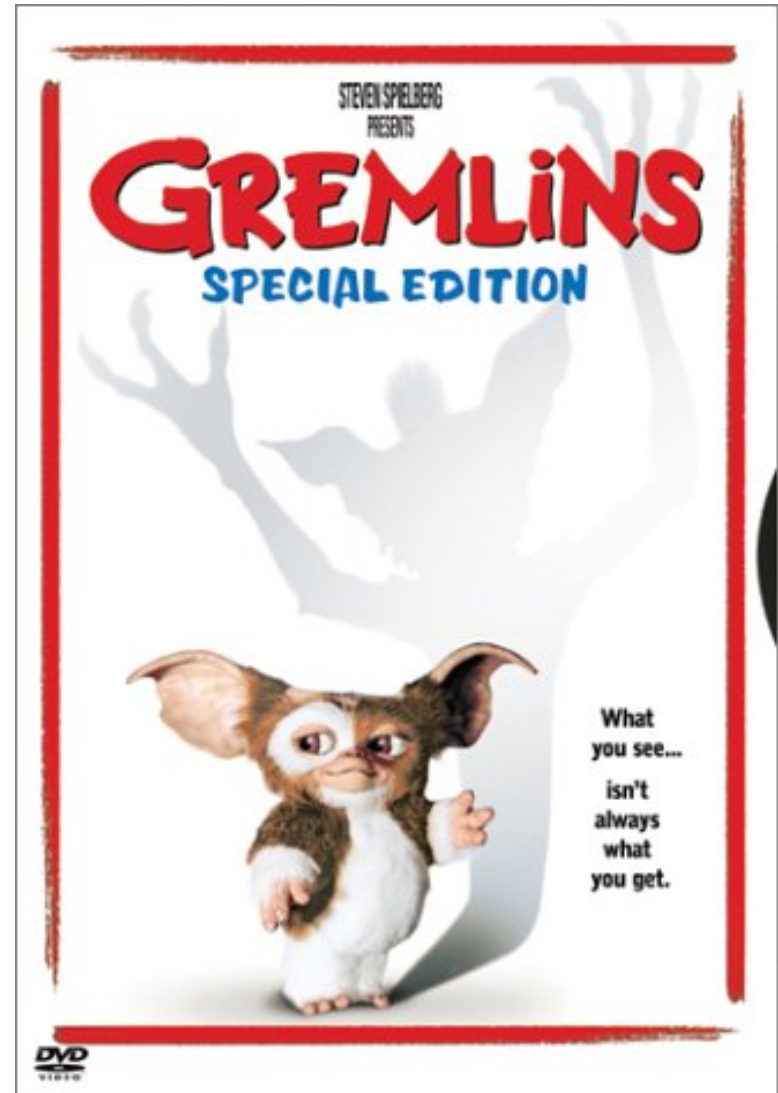
- This 1993 Mel Brooks parody film features "The Man in Black" as "Kevin Costner" and also stars Patrick Stewart as King Richard. It includes the exchange: *"And why would the people listen to you? / Because, unlike some other Robin Hoods, I can speak with an English accent."*

# Q: General (452 / 842)

- Name any 3 of the 22 letters in the Hebrew alphabet.



ENCYCLOPAEDIA OF THE ORIENT

| | | | | | |
|---|---|---|---|---|---|
| b | d | f | g | h | kh |
| k | l | m | n | o | oo |
| p | r | s | sh | t | |
| ts | v | y | z | no sound | |

HEBREW ALPHABET

- This 1965 **Wham-O** toy is an extremely elastic sphere made of a rubber polymer with a high coefficient of restitution. When dropped from shoulder level onto a hard surface it rebounds to about 90% of its original height.

# A:  Games  (489 / 842)

- **Super Ball**

  - *Trivia: When Lamar Hunt saw his daughter playing with a Super Ball, it inspired him to name the new AFL-NFL World Championship Game the Super Bowl.*

# Informal Semantics

- We consider only closed terms
- The evaluation of

$$(\lambda \; x. \; e) \; f$$

  – Binds x to f

  – Evaluates e *with the new binding*

  – Yields the result of this evaluation

- Like a function call, or like "let x = f in e"
- Example:

$$(\lambda \; f. \; f \; (f \; e)) \; g \quad \textit{evaluates to} \quad g \; (g \; e)$$

# Operational Semantics

- Many operational semantics for the $\lambda$-calculus
- All are based on the equation
$$(\lambda\ x.\ e)\ f =_\beta [f/x]e$$

  usually read from left to right
- This is called the <u>$\beta$-rule</u> and the evaluation step a <u>$\beta$-reduction</u>
- The subterm $(\lambda\ x.\ e)\ f$ is a <u>$\beta$-redex</u>
- We write $e \to_\beta g$ to say that $e$ $\beta$-reduces to $g$ in one step
- We write $e \to_\beta^* g$ to say that $e$ $\beta$-reduces to $g$ in 0 or more steps
  - Remind you of the small-step opsem term rewriting?

# Examples of Evaluation

- The identity function:

$$(\lambda\ x.\ x)\ E \to [E\ /\ x]\ x = E$$

- Another example with the identity:

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

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

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

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

$[\lambda\ y.\ y\ /\ x]\ x = \lambda\ y.\ y$

- A *non-terminating* evaluation:

$(\lambda\ x.\ xx)\ (\lambda\ y.\ yy) \to$

$[\lambda\ y.\ yy\ /\ x]\ xx = (\lambda\ y.\ yy)\ (\lambda\ y.\ yy) \to \dots$

- Try T T, where T = $\lambda x.\ x\ x\ x$



TAIL RECURSION

# Evaluation and the Static Scope

- The definition of substitution guarantees that evaluation respets static scoping:

$$(\lambda x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \rightarrow_\beta \lambda z.\ z\ (y\ (\lambda v.\ v))$$
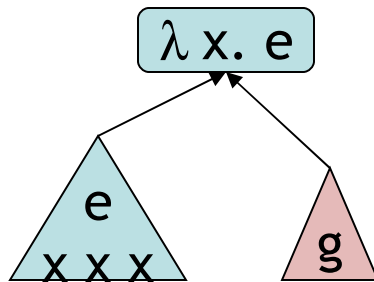
(**y** remains free, i.e., defined externally)

- If we forget to rename the bound y:

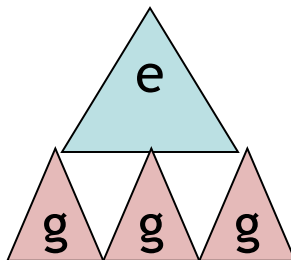$$(\lambda x.\ (\lambda y.\ y\ x))\ (y\ (\lambda x.\ x)) \rightarrow_\beta^* \lambda y.\ y\ (y\ (\lambda v.\ v))$$

(**y** was free before but *is bound now*)

# Another View of Reduction

- The application



$\lambda\,x.\ e$

e
x x x

g

- Becomes:

e

g   g   g

(terms can grow substantially through β-reduction!)

# Normal Forms

- A term without redexes is in <u>normal form</u>
- A reduction sequence stops at a normal form

- If $e$ is in normal form and $e \rightarrow_\beta^* f$ then $e$ is identical to $f$

- $K = \lambda x. \lambda y. x$ is in normal form
- $K\ I$ is *not* in normal form

# Nondeterministic Evaluation

- We define a small-step reduction relation

$$\frac{}{(\lambda x.\ e)\ f \rightarrow [f/x]e}$$

$$\frac{e_1 \rightarrow e_2}{e_1\ f \rightarrow e_2\ f} \qquad \frac{f_1 \rightarrow f_2}{e\ f_1 \rightarrow e\ f_2}$$

$$\frac{e \rightarrow f}{\lambda x.\ e \rightarrow \lambda x.\ f}$$

- This is a non-deterministic semantics
- Note that we evaluate under $\lambda$ *(where?)*

# Lambda Calculus Contexts

- Define contexts with one hole

- H ::= $\bullet$ | $\lambda$ x. H | H e | e H

- Write H[e] to denote the filling of the hole in H with the expression e

- Example:

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

- Filling the hole allows variable capture!

    H = $\lambda$ x. x $\bullet$      H[x] = $\lambda$ x. x x

# Contextual Opsem

$$(\lambda x.\ e)\ f \to [f/x]e$$
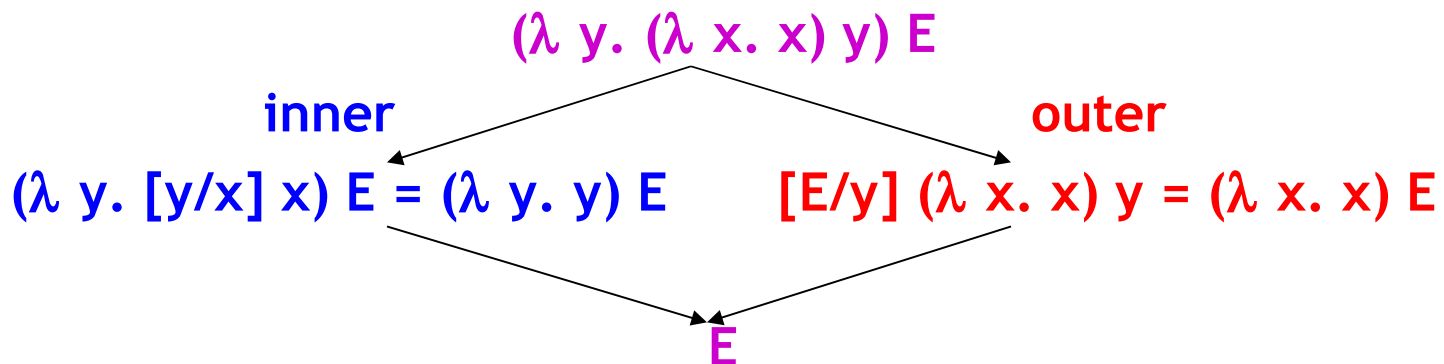
$$\frac{e \to f}{H[e] \to H[f]}$$

- Contexts allow concise formulations of <u>congruence</u> rules (application of local reduction rules on subterms)

- Reduction occurs at a $\beta$-redex that can be anywhere inside the expression

- The latter rule is called a <u>congruence</u> or structural rule

- The above rules to not specify which redex must be reduced first

# The Order of Evaluation

- In a $\lambda$-term there could be more than one instance of $(\lambda$ x. e) f, as in:
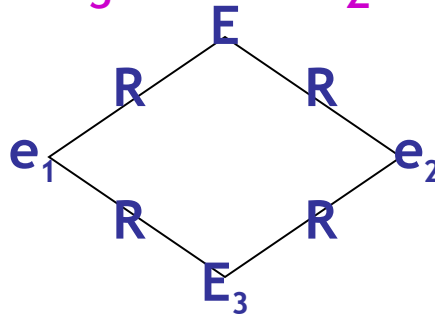
$$(\lambda \text{ y. } (\lambda \text{ x. x) y) E}$$

  – Could reduce the inner or outer $\lambda$

  – Which one should we pick?

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

inner                      outer

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

E

# The Diamond Property

- A relation $R$ has the <u>diamond property</u> if whenever $e\ R\ e_1$ and $e\ R\ e_2$ then there exists $e_3$ such that $e_1\ R\ e_3$ and $e_2\ R\ e_3$



- $\rightarrow_\beta$ does *not* have the diamond property

- $\rightarrow_\beta^*$ has the diamond property

- Also called the <u>confluence property</u>

# A Diamond In The Rough

- Languages defined by non-deterministic sets of rules are <span style="color:red">common</span>
  - Logic programming languages
  - Expert systems
  - Constraint satisfaction systems
    - And thus most pointer analyses ...
  - Dataflow systems
  - Makefiles
- It is useful to know whether such systems have the diamond property

# (Beta) Equality

- Let $=_\beta$ be the reflexive, transitive and **symmetric** closure of $\to_\beta$

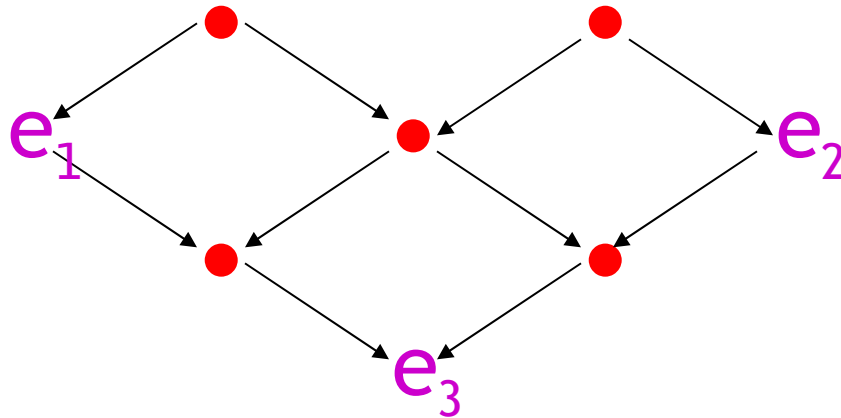$$=_\beta \text{ is } (\to_\beta \cup \leftarrow_\beta)^*$$

- That is, $e =_\beta f$ if $e$ converts to $f$ via a sequence of forward and backward $\to_\beta$
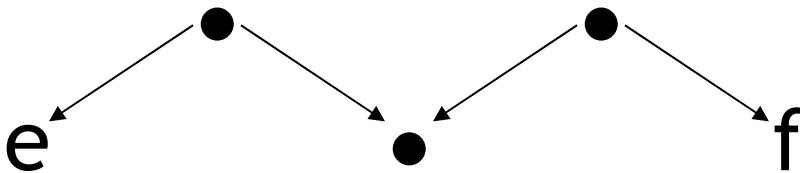
# The Church-Rosser Theorem

- If $e_1 =_\beta e_2$ then there exists $e_3$ such that $e_1 \rightarrow_\beta^* e_3$ and $e_2 \rightarrow_\beta^* e_3$



- Proof (informal): apply the diamond property as many times as necessary

# Corollaries

- If $e_1 =_\beta e_2$ and $e_1$ and $e_2$ are normal forms then $e_1$ is identical to $e_2$

  – From C-R we have $\exists e_3.\ e_1 \rightarrow_\beta^* e_3$ and $e_2 \rightarrow_\beta^* e_3$

  – Since $e_1$ and $e_2$ are normal forms they are identical to $e_3$

- If $e \rightarrow_\beta^* e_1$ and $e \rightarrow_\beta^* e_2$ and $e_1$ and $e_2$ are normal forms then $e_1$ is identical to $e_2$

  – "All terms have a unique normal form."

# Evaluation Strategies

- Church-Rosser theorem says that independent of the reduction strategy we will find $\leq 1$ normal form

- But some reduction strategies might find 0

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

  $\qquad (\lambda x.\ z)\ ((\lambda y.\ y\ y)\ (\lambda y.\ y\ y)) \rightarrow \ldots$

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

- There are three traditional strategies
  - normal order  (never used, always works)
  - call-by-name  (rarely used, cf. TeX)
  - call-by-value  (amazingly popular)

# Civilization: Call By Value

- Normal Order
  - Evaluates the left-most redex not contained in another redex
  - If there is a normal form, this finds it
  - Not used in practice: requires partially evaluating function pointers and looking "inside" functions
- Call-By-Name ("lazy")
  - Don't reduce under $\lambda$, don't evaluate a function argument (until you need to)
  - Does not always evaluate to a normal form
- **Call-By-Value ("eager" or "strict")**
  - Don't reduce under $\lambda$, *do* evaluate a function's argument right away
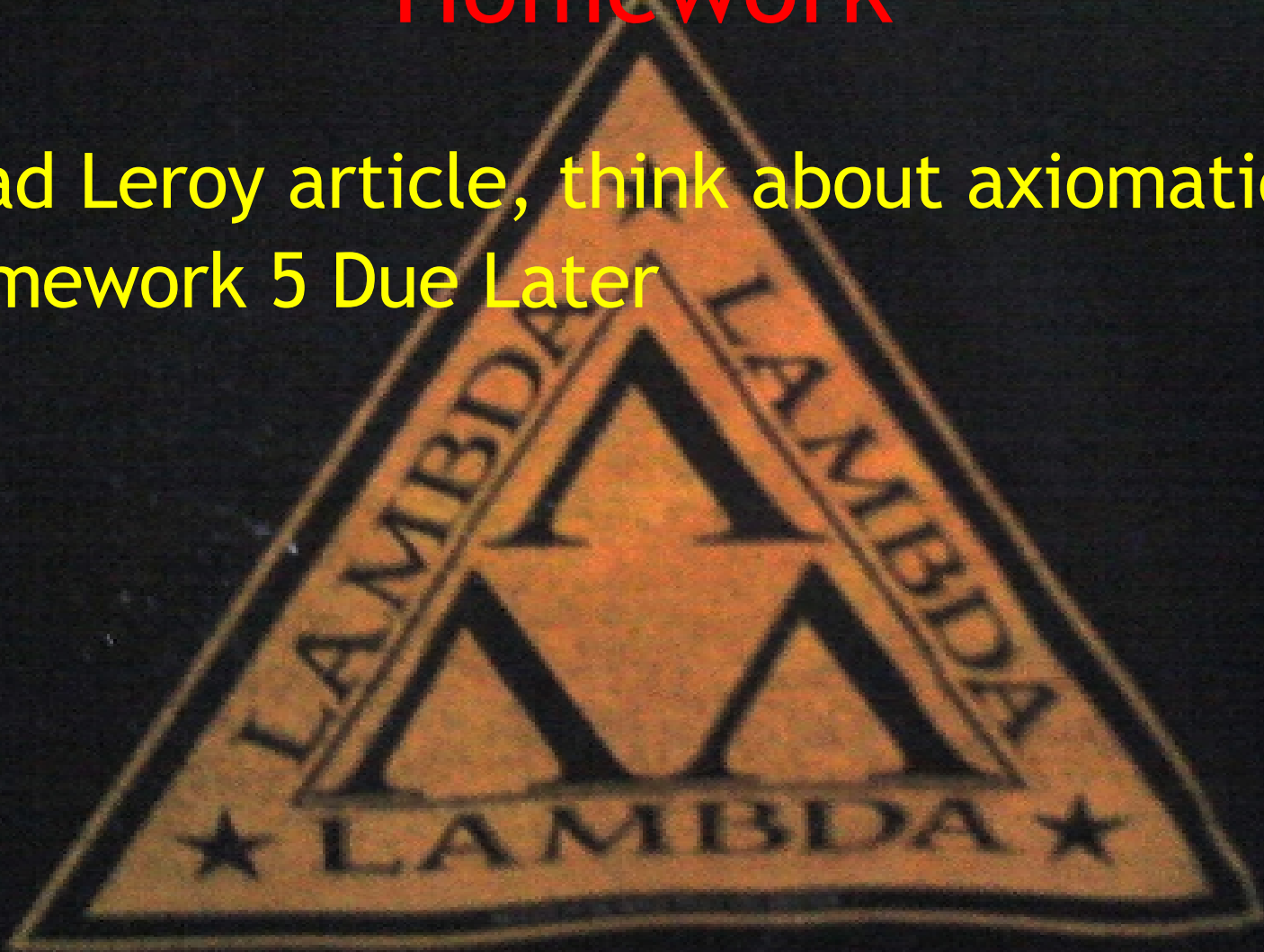  - Finds normal forms less often than the other two

# Endgame

- This time: $\lambda$ syntax, semantics, reductions, equality, ...

- Next time: encodings, real prorams, type systems, and all the fun stuff!

*Wisely done, Mr. Freeman. I will see you up ahead.*

# Homework

- Read Leroy article, think about axiomatic
- Homework 5 Due Later

# Tricksy On The Board Answer

- Is this rule unsound?

$$\frac{\vdash \{A \wedge p\}\ c_{then}\ \{B_{then}\} \qquad \vdash \{A \wedge \neg p\}\ c_{else}\ \{B_{else}\}}{\vdash \{A\}\ \text{if } p \text{ then } c_{then} \text{ else } c_{else}\ \{B_{then} \vee B_{else}\}}$$

- Nope: it's our basic rule plus 2x consequence

$$\frac{\vdash \{A \wedge p\}\ c_1\ \{B\} \qquad \vdash \{A \wedge \neg p\}\ c_2\ \{B\}}{\vdash \{A\}\ \text{if } p \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

$$\frac{\vdash A' \Rightarrow A \quad \vdash \{A\}\ c\ \{B\} \quad \vdash B \Rightarrow B'}{\vdash \{A'\}\ c\ \{B'\}}$$

- Note that $B_{then} \Rightarrow B_{then} \vee B_{else}$