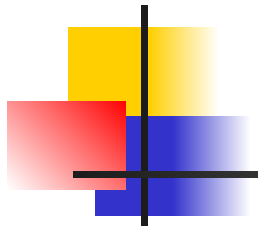


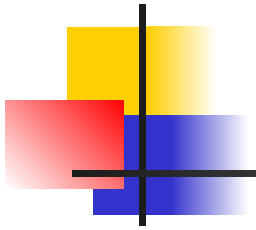
Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

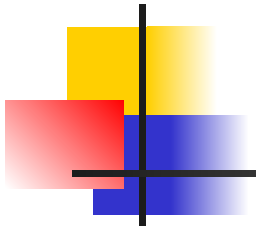
https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



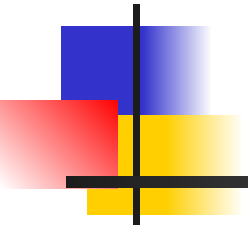
Lambda Calculus - Motivation

- A **beautiful**, **simple**, **Turing-complete** programming language
- Captures the essence of **functions**, function **application**, and **evaluation**
- Serves as a **theory of computation**
- Extremely **elegant** and useful for reasoning
- Two kinds: **untyped** and **typed**

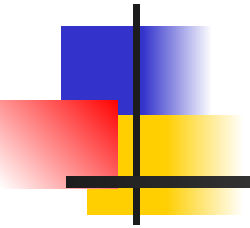


Lambda Calculus - Motivation

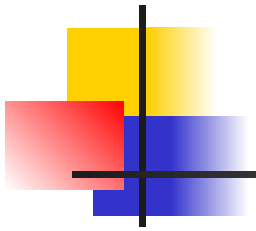
- A **beautiful**, **simple**, **Turing-complete** programming language
- Captures the essence of **functions**, function **application**, and **evaluation**
- Serves as a **theory of computation**
- Extremely **elegant** and useful for reasoning
- Two kinds: **untyped** and **typed**



Questions before we start?



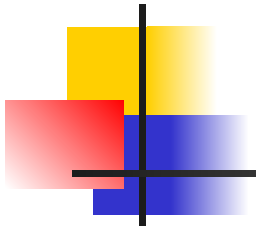
Untyped Lambda Calculus



Untyped λ-Calculus is All You Need

- **All you need:**
  - **Variables**: x, y, z, w, …
  - **Abstraction**: λ x . e

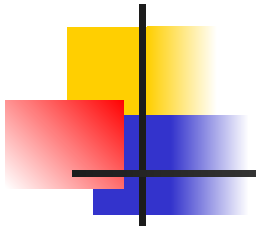
    (Function creation, think fun x -> e)
  - **Application**: e_1 e_2
  - **Grouping**: (e)
- With some environment, and some canonical values, this is **Turing-Complete!**



Untyped λ-Calculus is All You Need

- **All you need:**
  - **Variables**: x, y, z, w, …
  - **Abstraction**: λ x . e

    (Function creation, think fun x -> e)
  - **Application**: e_1 e_2
  - **Grouping**: (e)
- With some environment, and some canonical values, this is **Turing-Complete!**



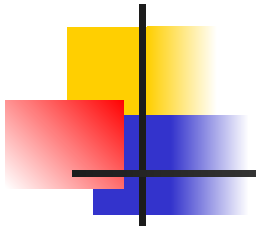
Untyped λ-Calculus Grammar

Formal BNF Grammar:

<expression> ::=
  | <variable>
  | <abstraction>
  | <application>
  | (<expression>)

<abstraction> ::= λ<variable>.<expression>

<application> ::= <expression> <expression>



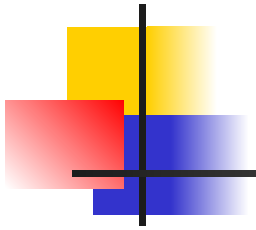
Untyped λ-Calculus Grammar

Formal BNF Grammar:

<expression> ::=
  | <variable>
  | <abstraction>
  | <application>
  | (<expression>)

<abstraction> ::= λ<variable>.<expression>

<application> ::= <expression> <expression>



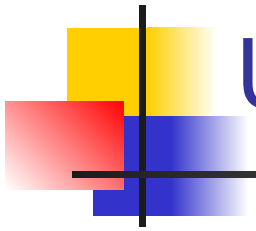
Untyped λ-Calculus Grammar

Formal BNF Grammar:

<expression> ::=
  | <variable>
  | <abstraction>
  | <application>
  | (<expression>)

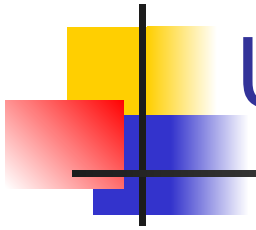
<abstraction> ::= λ<variable>.<expression>

<application> ::= <expression> <expression>



Untyped λ-Calculus Terminology

- **Variable binding**: λ x . e is a binding of x in e
- **Occurrence**: a location of a subterm in a term
  - **Bound occurrence**: occurrences of x in λ x . e
  - **Free occurrence**: one that is not bound
- **Scope of binding**: in λ x. e, all occurrences in e not in a subterm of the form λ x. e' (same x)
- **Free variables**: all variables having free occurrences in a term



Untyped λ-Calculus Terminology

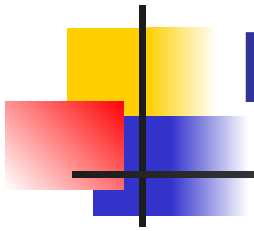
- **Variable binding**: λ x . e is a binding of x in e
- **Occurrence**: a location of a subterm in a term
  - **Bound occurrence**: occurrences of x in λ x . e
  - **Free occurrence**: one that is not bound
- **Scope of binding**: in λ x. e, all occurrences in e not in a subterm of the form λ x. e' (same x)
- **Free variables**: all variables having free occurrences in a term

Untyped λ-Calculus Terminology

- **Variable binding**: λ x . e is a binding of x in e
- **Occurrence**: a location of a subterm in a term
  - **Bound occurrence**: occurrences of x in λ x . e
  - **Free occurrence**: one that is not bound
- **Scope of binding**: in λ x. e, all occurrences in e not in a subterm of the form λ x. e′ (same x)
- **Free variables**: all variables having free occurrences in a term

Untyped λ-Calculus Terminology

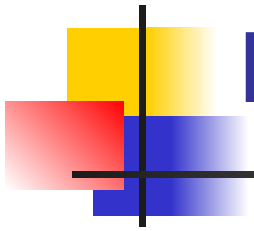
- **Variable binding**: λ x . e is a binding of x in e
- **Occurrence**: a location of a subterm in a term
  - **Bound occurrence**: occurrences of x in λ x . e
  - **Free occurrence**: one that is not bound
- **Scope of binding**: in λ x. e, all occurrences in e not in a subterm of the form λ x. e' (same x)
- **Free variables**: all variables having free occurrences in a term



Example

- **Label occurrences and scope:**

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

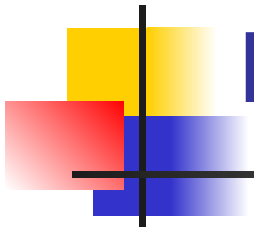


Example

- **Label occurrences and scope:**

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

$$1\quad 2\quad 3\quad 4\qquad 5\ 6\ 7\quad 8\quad 9$$



Example

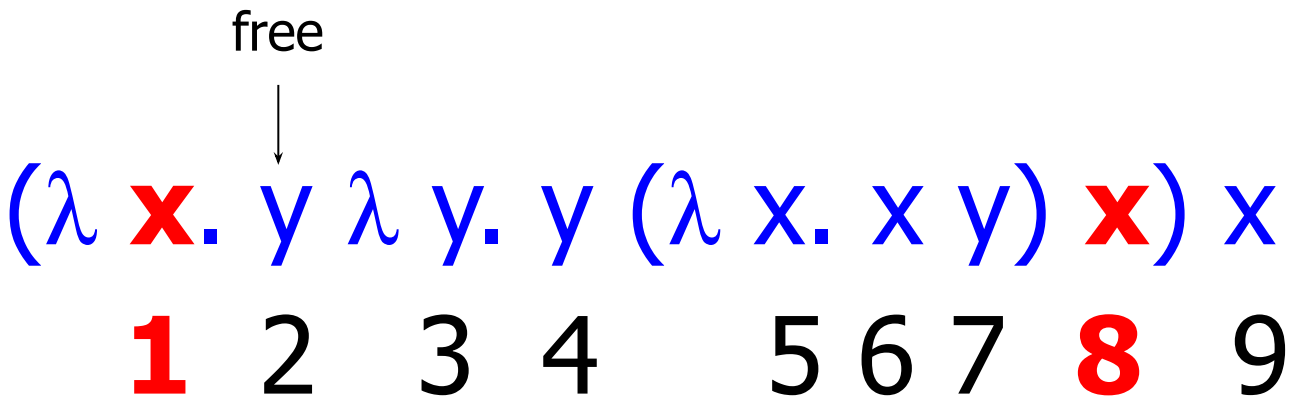
- **Label occurrences and scope:**

$$(\lambda\, \mathbf{x}.\ y\ \lambda\, y.\ y\ (\lambda\, x.\ x\ y)\ \mathbf{x})\ x$$

$$\mathbf{1}\ \ 2\ \ \ 3\ \ 4\ \ \ \ \ 5\ 6\ 7\ \ \mathbf{8}\ \ \ 9$$

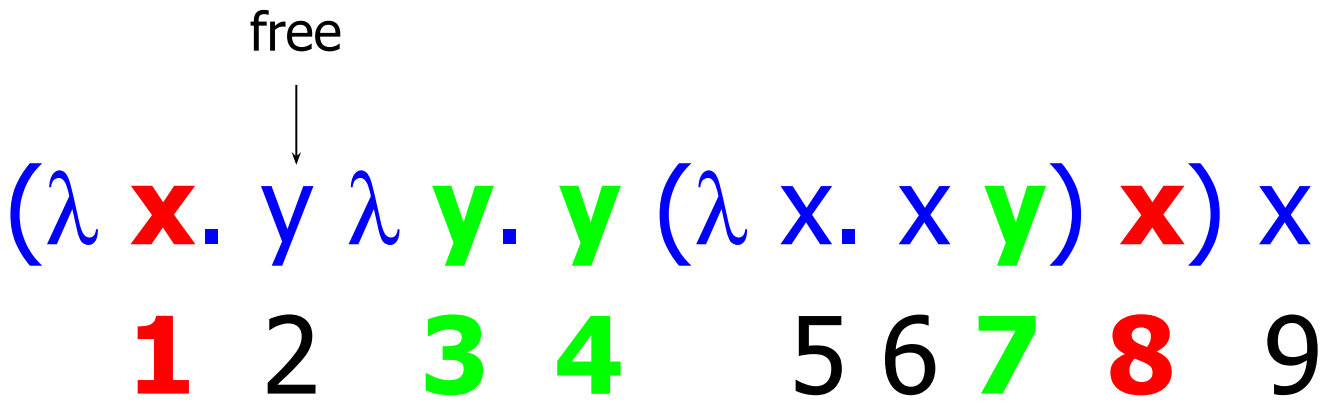
Example

- **Label occurrences and scope:**



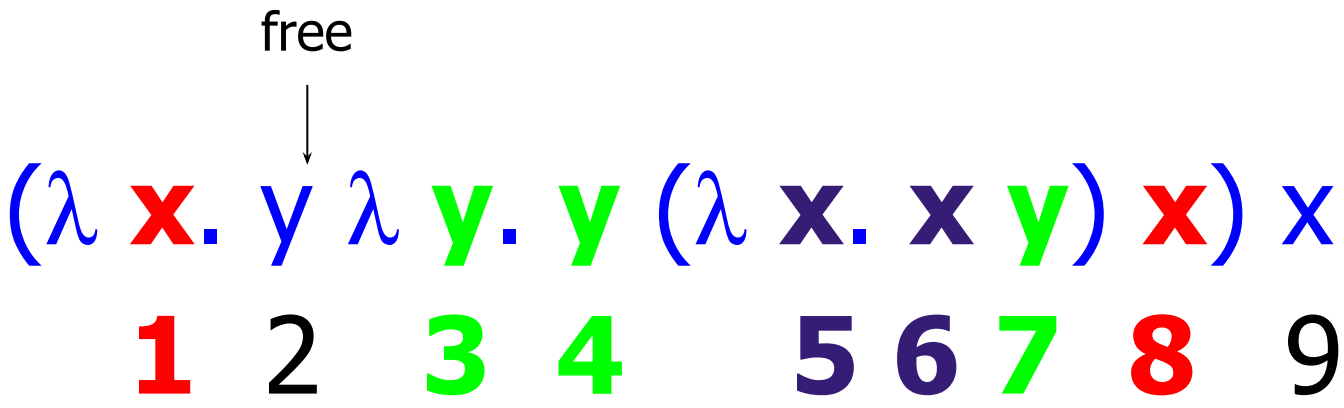
Example

- **Label occurrences and scope:**



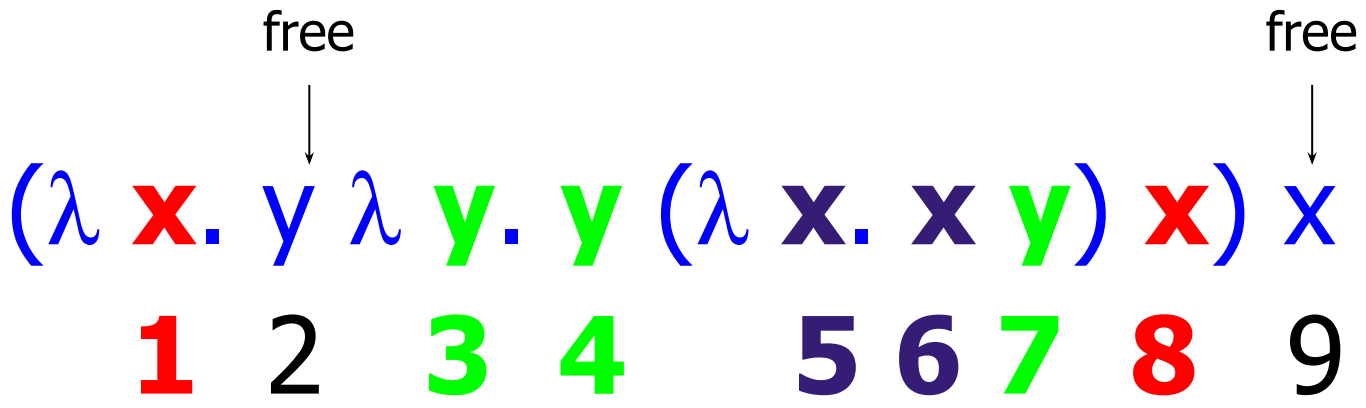
Example

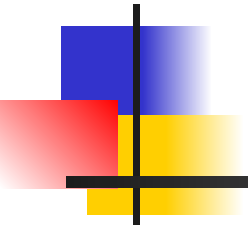
- **Label occurrences and scope:**



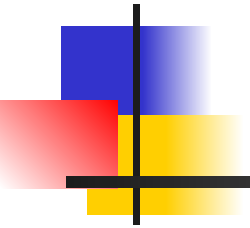
Example

- **Label occurrences and scope:**

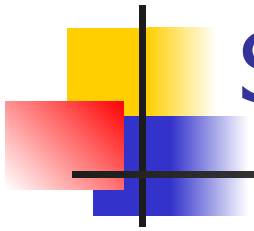




Questions so far?



Computation



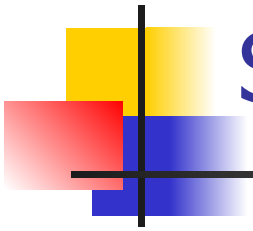
Some Intuition

Identity Function:

$(\lambda\ \mathbf{x}.\ \mathbf{x})$

Applying Identity Function:

$(\lambda\ \mathbf{x}.\ \mathbf{x})\ y \Rightarrow^* y$



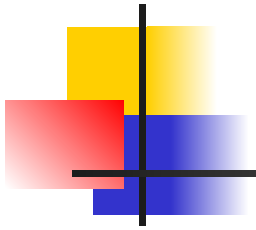
Some Intuition

Identity Function:

$(\lambda\ \mathbf{x}.\ \mathbf{x})$

Applying Identity Function:

$(\lambda\ \mathbf{x}.\ \mathbf{x})\ y \Rightarrow^*\ y$



Untyped λ-Calculus

- How do you **compute** with the λ-calculus?
- Roughly speaking, by **substitution**:

  $(\lambda\, x.\, e_1)\, e_2 \;\Rightarrow^*\; e_1\,[e_2 / x]$

  (modulo subtleties to deal with variables)

Transition Semantics for λ-Calculus

Lazy Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{}{(\lambda\ x\ .\ E)\ E' \rightarrow E[E'/x]}\ \textbf{L-App}$$

Computation

Transition Semantics for λ-Calculus

Eager Evaluation

$$\frac{E \rightarrow E''}{E\ E' \rightarrow E''\ E'}\ \textbf{App}$$

$$\frac{E' \rightarrow E''}{(\lambda\ x\ .\ E)\ E' \rightarrow (\lambda\ x\ .\ E)\ E''}\ \textbf{E-App}$$

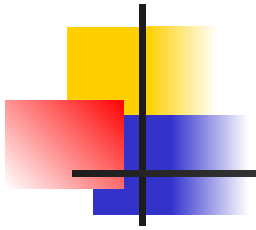
$$\frac{}{(\lambda\ x\ .\ E)\ V \rightarrow E[V/x]}\ \textbf{V-App}$$

Where V is a variable or abstraction (value)

Computation

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is **Turing Complete**
  - Can express any sequential computation
  - Yes, in that few computation rules
- But it'd suck to use as-is:
  - How to express basic **data**: booleans, integers, etc?
  - How to express **recursion**?
  - What about **constants**? **If-then-else**?
    - "Just" a convenience—can be added as syntactic sugar



How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is **Turing Complete**
  - Can express any sequential computation
  - Yes, in that few computation rules
- But it'd suck to use as-is:
  - How to express basic **data**: booleans, integers, etc?
  - How to express **recursion**?
  - What about **constants**? **If-then-else**?
    - "Just" a convenience—can be added as syntactic sugar

How Powerful is the Untyped λ-Calculus?

- The untyped λ-calculus is **Turing Complete**
  - Can express any sequential computation
  - Yes, in that few computation rules
- But it'd suck to use as-is:
  - How to express basic **data**: booleans, integers, etc?
  - How to express **recursion**?
  - What about **constants**? **If-then-else**?
    - "Just" a convenience—can be added as syntactic sugar

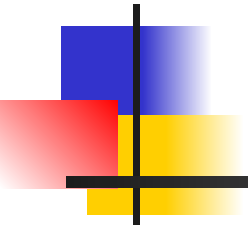
Clever encodings do exist

How Powerful is the **Typed** λ-Calculus?

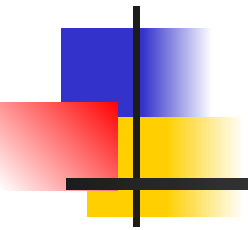
- **Sometimes Not Turing Complete**
- Depends on the type system!
  - Types rule out invalid programs
  - What is the type of (f f)?
  - Types are *not* syntactic sugar! They disallow some terms
- e.g., simply typed λ-calculus is less powerful than the untyped λ-Calculus: **not Turing Complete** (no recursion)

How Powerful is the **Typed** λ-Calculus?

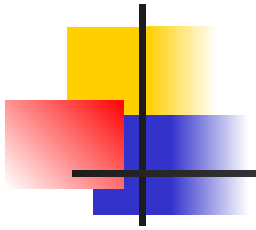
- **Sometimes Not Turing Complete**
- Depends on the type system!
  - Types rule out invalid programs
  - What is the type of (f f)?
  - Types are *not* syntactic sugar! They disallow some terms
- e.g., simply typed λ-calculus is less powerful than the untyped λ-Calculus: **not Turing Complete** (no recursion)



Questions so far?

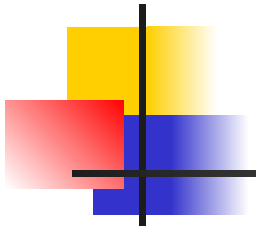


Normalization: Another way to think about meaning



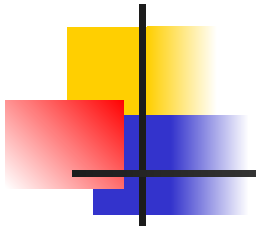
Equality

- A problem that shows up *everywhere*: How do you tell if two terms in this language are "the same" as each other?
  - **Compute** them all the way, then see if the **result** is the same
  - Want some way of **normalizing** the terms—choose some **normal form**
  - "Same" means **same normal form**
- Typically a simple, syntactic notion of equality



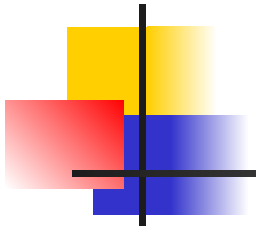
Equality

- A problem that shows up *everywhere*: How do you tell if two terms in this language are "the same" as each other?
  - **Compute** them all the way, then see if the **result** is the same
  - Want some way of **normalizing** the terms—choose some **normal form**
  - "Same" means **same normal form**
- Typically a simple, syntactic notion of equality



Equality

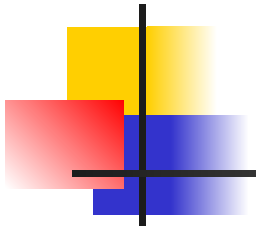
- A problem that shows up *everywhere*: How do you tell if two terms in this language are "the same" as each other?
  - **Compute** them all the way, then see if the **result** is the same
  - Want some way of **normalizing** the terms—choose some **normal form**
  - "Same" means **same normal form**
- Typically a simple, syntactic notion of equality



Equality

- Programming languages researchers really like Greek letters for some reason
- So when we define computation rules to get terms into their normal forms, we name them after Greek letters:
  - **α-conversion**: renaming variables
  - **β-reduction**: reducing function application
- **Equality** of lambda terms in untyped lambda calculus is **αβ-equivalence**

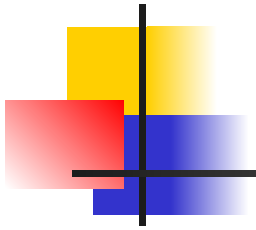
Normalization



Equality

- Programming languages researchers really like Greek letters for some reason
- So when we define computation rules to get terms into their normal forms, we name them after Greek letters:
  - **α-conversion**: renaming variables
  - **β-reduction**: reducing function application
- **Equality** of lambda terms in untyped lambda calculus is **αβ-equivalence**

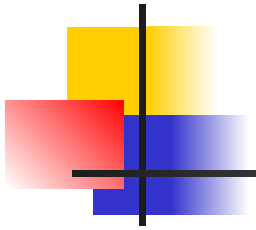
Normalization



Equality

- Programming languages researchers really like Greek letters for some reason
- So when we define computation rules to get terms into their normal forms, we name them after Greek letters:
  - **α-conversion**: renaming variables
  - **β-reduction**: reducing function application
- **Equality** of lambda terms in untyped lambda calculus is **αβ-equivalence**

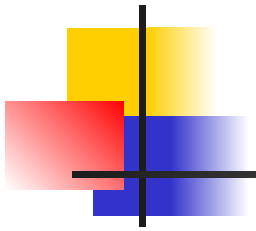
Normalization



Equality

- Programming languages researchers really like Greek letters for some reason
- So when we define computation rules to get terms into their normal forms, we name them after Greek letters:
  - **α-conversion**: renaming variables
  - **β-reduction**: reducing function application
- **Equality** of lambda terms in untyped lambda calculus is **αβ-equivalence**

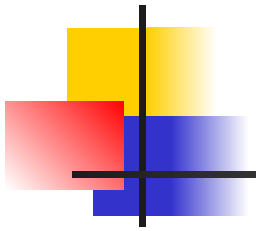
Normalization



α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y



α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:

λ x. x y ~~–α–>~~ λ y. y y

α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:

y is free in body on LHS, but not in body on RHS

λ x. x **y** –α–> λ y. y y

α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:

y is free in body on LHS, but not in body on RHS

λ x. x **y** ~~–α–>~~ λ **y**. **y y**

α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

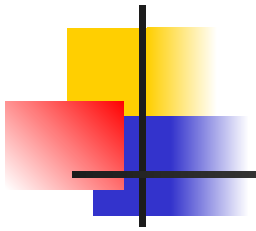
1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:

y is free in body on LHS, but not in body on RHS

λ x. x **y** –α–> λ **y**. **y y**

Can't just rename x to y; get something different



α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:

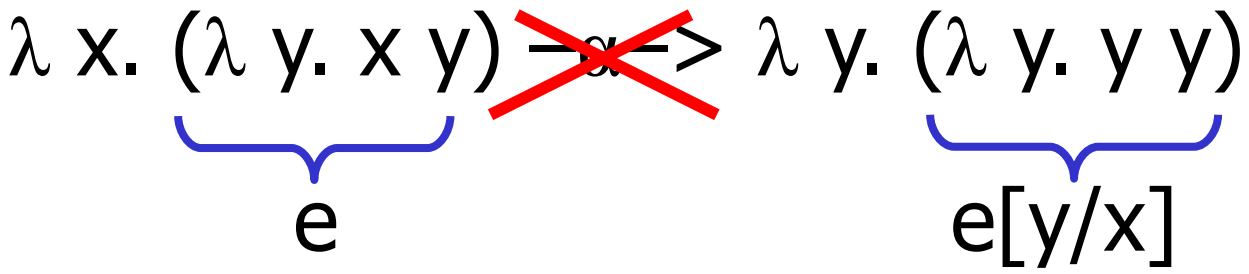
λ x. (λ y. x y) ~~–α–>~~ λ y. (λ y. y y)

α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:



Normalization

α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Bad:

Free occurrence of x in body on LHS becomes bound in wrong way when replaced by y on RHS

λ **x**. (λ y. **x** y) ~~–α–>~~ λ y. (λ y. y y)

e e[y/x]

Normalization

α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

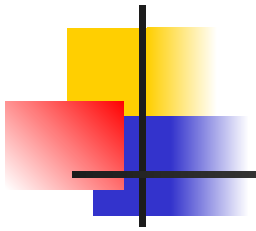
Bad:

Free occurrence of x in body on LHS becomes bound in wrong way when replaced by y on RHS

λ **x**. (λ y. **x** y) ~~–α–>~~ λ y. (λ **y**. **y** y)

e e[y/x]

Normalization



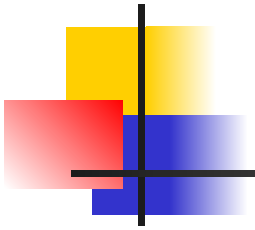
α-Conversion

λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Good:

λ x. (λ y. y) x –α–> λ y. (λ y. y) y



α-Conversion

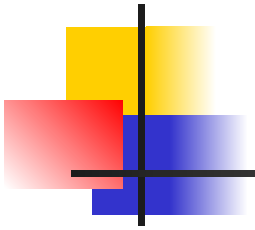
λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Good:

Renaming x on LHS to y on RHS doesn't change the meaning!

λ x. (λ y. y) x –α–> λ y. (λ y. y) y



α-Conversion

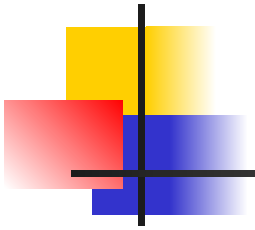
λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Good:

Renaming x on LHS to y on RHS doesn't change the meaning!

λ **x**. (λ y. y) **x** –α–> λ y. (λ y. y) y



α-Conversion

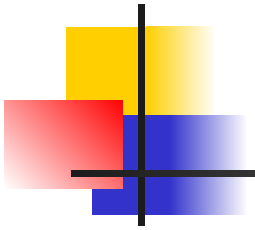
λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

Good:

Renaming x on LHS to y on RHS doesn't change the meaning!

λ **x**. (λ y. y) **x** –α–> λ **y**. (λ y. y) **y**



α-Conversion

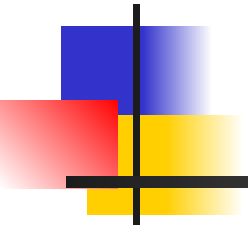
λ x. e –α–> λ y. (e [y/x]) provided that:

1. y is not free in e
2. No free occurrence of x in e becomes bound in e when replaced by y

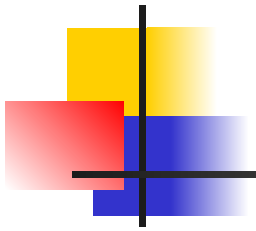
Good:

Renaming first y on LHS to x on RHS doesn't change the meaning!

λ **y**. (λ y. y) **y** –α–> λ **x**. (λ y. y) **x**

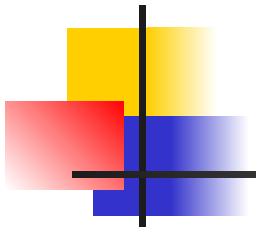


Questions so far?



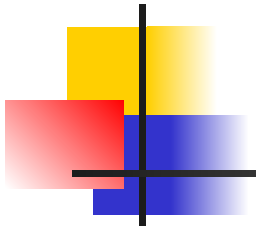
α-Equivalence

- **α-equivalence** is the **smallest congruence** containing **α-conversion**
- Let ~ be a relation on lambda terms. ~ is a **congruence** if
  - it is an equivalence relation, and
  - if $e_1 \sim e_2$ then
    - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
    - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$
- One usually treats α-equivalent terms as equal, i.e., uses α-equivalence classes of terms



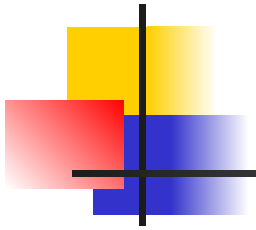
α-Equivalence

- **α-equivalence** is the **smallest congruence** containing **α-conversion**
- Let ~ be a relation on lambda terms. ~ is a **congruence** if
  - it is an equivalence relation, and
  - if $e_1 \sim e_2$ then
    - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
    - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$
- One usually treats α-equivalent terms as equal, i.e., uses α-equivalence classes of terms



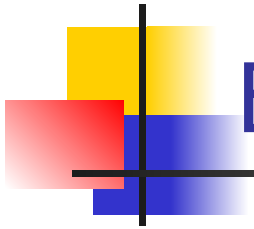
α-Equivalence

- **α-equivalence** is the **smallest congruence** containing **α-conversion**
- Let ~ be a relation on lambda terms. ~ is a **congruence** if
  - it is an equivalence relation, and
  - if $e_1 \sim e_2$ then
    - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
    - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$
- One usually treats α-equivalent terms as equal, i.e., uses α-equivalence classes of terms



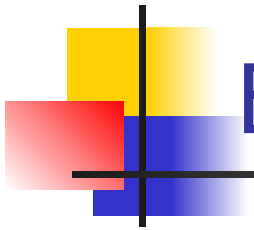
α-Equivalence

- **α-equivalence** is the **smallest congruence** containing **α-conversion**
- Let ~ be a relation on lambda terms. ~ is a **congruence** if
  - it is an equivalence relation, and
  - if $e_1 \sim e_2$ then
    - $(e\ e_1) \sim (e\ e_2)$ and $(e_1 e) \sim (e_2\ e)$
    - $\lambda\ x.\ e_1 \sim \lambda\ x.\ e_2$
- One usually treats α-equivalent terms as equal, i.e., uses α-equivalence classes of terms



Example

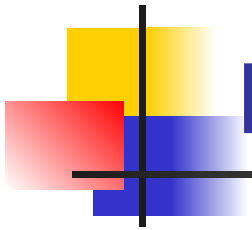
Show: $\lambda\, x.\ (\lambda\, y.\ y\ x)\ x \sim\alpha\sim \lambda\, y.\ (\lambda\, x.\ x\ y)\ y$



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

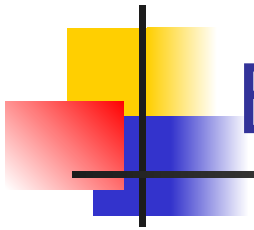
λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

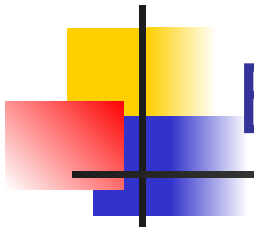
λ **x**. (λ y. y x) **x** –α–> λ **z**. (λ y. y **z**) **z**



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

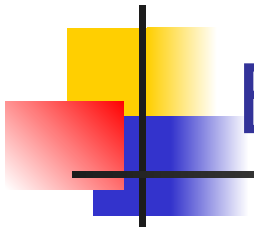
λ **x**. (λ y. y x) **x** –α–> λ **z**. (λ y. y **z**) **z** so
λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

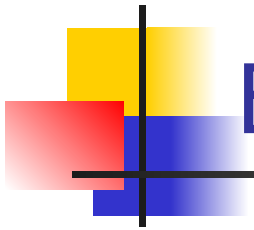
λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x **~α~** λ z. (λ y. y z) z



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x ~α~ λ z. **(λ y. y z)** z

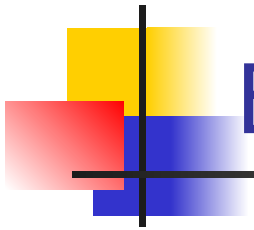


Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x ~α~ λ z. **(λ y. y z)** z

(λ y. y z) –α–> (λ x. x z)



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z

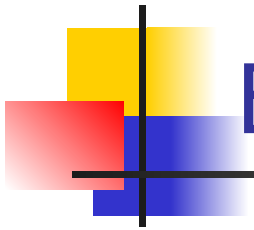
(λ **y. y** z) –α–> (λ **x. x** z)

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z)\ \mathbf{\sim\alpha\sim}\ (\lambda\ x.\ x\ z)$



Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x ~α~ λ z. **(λ y. y z) z**

(λ y. y z) –α–> (λ x. x z) so
(λ y. y z) ~α~ (λ x. x z)

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ z.\ \mathbf{(\lambda\ y.\ y\ z)\ z}$

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
$\mathbf{(\lambda\ y.\ y\ z)\ z} \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$

Example

Show: λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y

λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z

(λ y. y z) –α–> (λ x. x z) so
(λ y. y z) ~α~ (λ x. x z) so
(λ y. y z) **z** ~α~ (λ x. x z) **z**

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim$ **$\lambda\ z.\ (\lambda\ y.\ y\ z)\ z$**

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z)\ z \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim$ **$\lambda\ z.\ (\lambda\ y.\ y\ z)\ z$**

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z)\ z \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$ so
$\lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ $\sim\alpha\sim \lambda\ z.\ (\lambda\ x.\ x\ z)\ z$

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z)\ z \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$ so
$\lambda\ \mathbf{z.}\ (\lambda\ y.\ y\ z)\ z \sim\alpha\sim \lambda\ \mathbf{z.}\ (\lambda\ x.\ x\ z)\ z$

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z)\ z \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$ so
$\lambda\ z.\ (\lambda\ y.\ y\ z)\ z \sim\alpha\sim \lambda\ \mathbf{z.\ (\lambda\ x.\ x\ z)\ z}$

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so
$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so
$(\lambda\ y.\ y\ z)\ z \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$ so
$\lambda\ z.\ (\lambda\ y.\ y\ z)\ z \sim\alpha\sim \lambda\ \mathbf{z.\ (\lambda\ x.\ x\ z)\ z}$

$\lambda\ \mathbf{z.\ (\lambda\ x.\ x\ z)\ z} \text{ –}\alpha\text{–>} \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$ so
$\lambda\ z.\ (\lambda\ x.\ x\ z)\ z \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

Example

Show: $\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \sim\alpha\sim \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$

$\lambda\ x.\ (\lambda\ y.\ y\ x)\ x \text{ –}\alpha\text{–>} \lambda\ z.\ (\lambda\ y.\ y\ z)\ z$ so

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

$(\lambda\ y.\ y\ z) \text{ –}\alpha\text{–>} (\lambda\ x.\ x\ z)$ so

$(\lambda\ y.\ y\ z) \sim\alpha\sim (\lambda\ x.\ x\ z)$ so

$(\lambda\ y.\ y\ z)\ z \sim\alpha\sim (\lambda\ x.\ x\ z)\ z$ so

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

$\lambda\ z.\ (\lambda\ x.\ x\ z)\ z \text{ –}\alpha\text{–>} \lambda\ y.\ (\lambda\ x.\ x\ y)\ y$ so

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

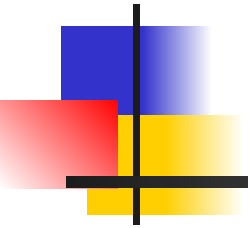
Example

Show: **λ x. (λ y. y x) x ~α~ λ y. (λ x. x y) y**

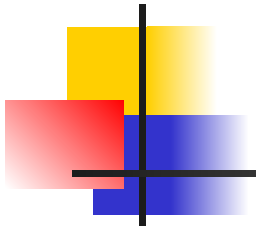
λ x. (λ y. y x) x –α–> λ z. (λ y. y z) z so
λ x. (λ y. y x) x ~α~ λ z. (λ y. y z) z

(λ y. y z) –α–> (λ x. x z) so
(λ y. y z) ~α~ (λ x. x z) so
(λ y. y z) z ~α~ (λ x. x z) z so
λ z. (λ y. y z) z ~α~ λ z. (λ x. x z) z

λ z. (λ x. x z) z –α–> λ y. (λ x. x y) y so
λ z. (λ x. x z) z ~α~ λ y. (λ x. x y) y

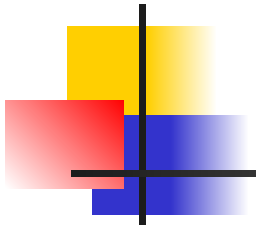


Questions so far?



Equality

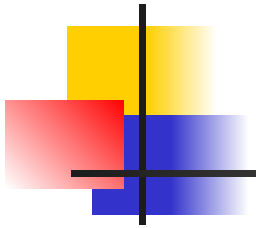
- Programming languages researchers really like Greek letters for some reason
- So when we define computation rules to get terms into their normal forms, we name them after Greek letters:
  - **α-conversion**: renaming variables
  - **β-reduction**: reducing function application
- **Equality** of lambda terms in untyped lambda calculus is **αβ-equivalence**



β-reduction

$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

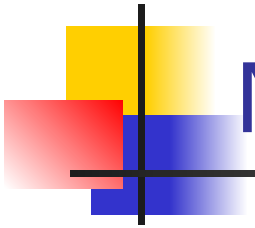
- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms



β-reduction

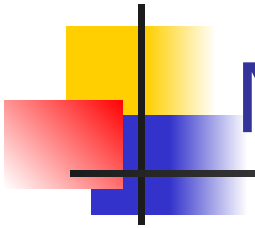
$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms



Needed: Substitution

- Defined on **α-equivalence classes** of terms
- e [r / x] means **replace every free occurrence** of x in e by r
  - e called **redex**; r called **residue**
- Provided that no variable free in e becomes bound in e [r / x]
  - **Rename** bound variables in e to **avoid capturing** free variables of r



Needed: Substitution

- Defined on **α-equivalence classes** of terms
- e [r / x] means **replace every free occurrence** of x in e by r
  - e called **redex**; r called **residue**
- Provided that no variable free in e becomes bound in e [r / x]
  - **Rename** bound variables in e to **avoid capturing** free variables of r

Needed: Substitution

We can define by cases:

- **Variable:**

  $x\,[r / x] = r$

  $y\,[r / x] = y$ if $y \neq x$

- **Application:**

  $(e_1\ e_2)\,[r / x] = ((e_1\,[r / x])\ (e_2\,[r / x]))$

- **Function:**

  $(\lambda\ x.\ e)\,[r / x] = (\lambda\ x.\ e)$

  $(\lambda\ y.\ e)\,[r / x] = \lambda\ y.\ (e\,[r / x])$ if $y \neq x$ and y not free in r

Needed: Substitution

We can define by cases:

- **Variable:**

  $x\,[r / x] = r$

  $y\,[r / x] = y$ if $y \neq x$

- **Application:**

  $(e_1\ e_2)\,[r / x] = ((e_1\,[r / x])\ (e_2\,[r / x]))$

- **Function:**

  $(\lambda\ x.\ e)\,[r / x] = (\lambda\ x.\ e)$

  $(\lambda\ y.\ e)\,[r / x] = \lambda\ y.\ (e\,[r / x])$ if $y \neq x$ and y not free in r

Needed: Substitution

We can define by cases:

- **Variable:**

  $x\,[r / x] = r$

  $y\,[r / x] = y$ if $y \neq x$

- **Application:**

  $(e_1\ e_2)\,[r / x] = ((e_1\,[r / x])\ (e_2\,[r / x]))$

- **Function:**

  $(\lambda\ x.\ e)\,[r / x] = (\lambda\ x.\ e)$

  $(\lambda\ y.\ e)\,[r / x] = \lambda\ y.\ (e\,[r / x])$ if $y \neq x$ and y not free in r

Normalization

Needed: Substitution

We can define by cases:

- **Variable:**

  $x\,[r / x] = r$

  $y\,[r / x] = y$ if $y \neq x$

- **Application:**

  $(e_1\ e_2)\,[r / x] = ((e_1\,[r / x])\ (e_2\,[r / x]))$

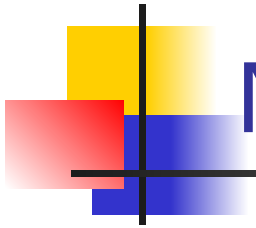
- **Function:**

  $(\lambda\ x.\ e)\,[r / x] = (\lambda\ x.\ e)$

  $(\lambda\ y.\ e)\,[r / x] = \lambda\ y.\ (e\,[r / x])$ if $y \neq x$ and y not free in r

Rename y in redex if needed to avoid capture

Normalization



Needed: Substitution

(λ y. y z) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] –α–>
  (λ w.w z) [(λ x. x y) / z] =
  λ w. w (λ x. x y)

Needed: Substitution

(λ **y**. y **z**) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] –α–>
  (λ w.w z) [(λ x. x y) / z] =
  λ w. w (λ x. x y)

Needed: Substitution

$(\lambda\ \mathbf{y}.\ y\ \mathbf{z})\ [(\lambda\ x.\ x\ \mathbf{y})\ /\ z] = ?$

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- $(\lambda\ y.\ y\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] \rightarrow_{\alpha}$
  $(\lambda\ w.w\ z)\ [(\lambda\ x.\ x\ y)\ /\ z] =$
  $\lambda\ w.\ w\ (\lambda\ x.\ x\ y)$

Needed: Substitution

(λ y. y z) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] –α–>
  (λ w.w z) [(λ x. x y) / z] =
  λ w. w (λ x. x y)

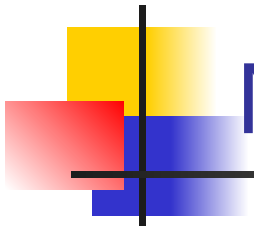
Rename y in redex if needed to avoid capture

Needed: Substitution

(λ y. y z) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] –α–>
  (λ w.w z) [(λ x. x y) / z] =
  λ w. w (λ x. x y)

Rename y in redex if needed to avoid capture



Needed: Substitution

(λ y. y z) [(λ x. x y) / z] = ?

- Problems?
  - z in redex in scope of y binding
  - y free in the residue
- (λ y. y z) [(λ x. x y) / z] –α–>
  (λ w.w z) [(λ x. x y) / z] =
  λ w. w (λ x. x y)

Then we can substitute

Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

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

  not

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

Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

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

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

  not

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

Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

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

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

  not

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

Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

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

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

  not

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

Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

  (λ y. y **z** (λ z. z)) [**(λ x. x)** / **z**] =

  λ y. y **(λ x. x)** (λ z. z)

  not

  λ y. y (λ x. x) (λ z. (λ x. x))

Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

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

  $\lambda\ y.\ y\ (\lambda\ \mathbf{x}.\ \mathbf{x})\ (\lambda\ \mathbf{z}.\ \mathbf{z})$

  not

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

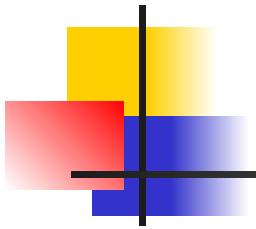
Needed: Substitution

- **Note**: only replace **free** occurrences
- e.g.,

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

  not

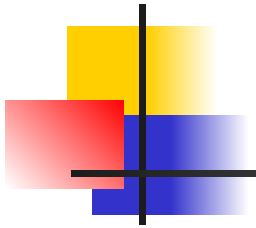
  $\lambda\ y.\ y\ \mathbf{(\lambda\ x.\ x)}\ (\lambda\ z.\ \mathbf{(\lambda\ x.\ x)})$



β-reduction

$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms



β-reduction

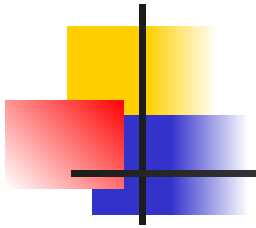
$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms

Example:

$(\lambda\ \mathbf{z}.\ (\lambda\ x.\ x\ y)\ \mathbf{z})\ (\lambda\ \mathbf{y}.\ \mathbf{y\ z}) \rightarrow_\beta$

??



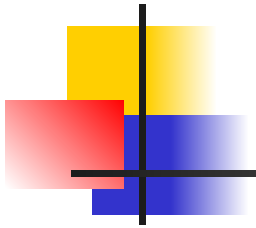
β-reduction

$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms

Example:

$(\lambda\ \mathbf{z}.\ (\lambda\ x.\ x\ y)\ \mathbf{z})\ \mathbf{(\lambda\ y.\ y\ z)} \rightarrow_\beta$
$(\lambda\ x.\ x\ y)\ \mathbf{(\lambda\ y.\ y\ z)}$



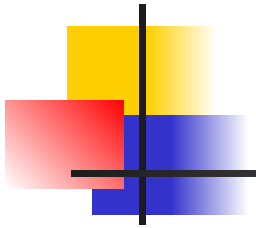
β-reduction

(λ x. e) r –β–> e [r / x]

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms

Example:

(λ **z**. (λ x. x y) **z**) **(λ y. y z)** –β–>
(λ **x**. **x** y) **(λ y. y z)** –β–>
??



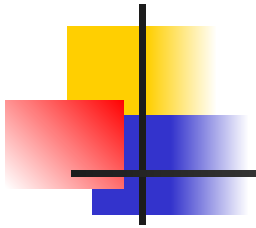
β-reduction

$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms

Example:

$(\lambda\ \mathbf{z}.\ (\lambda\ x.\ x\ y)\ \mathbf{z})\ \mathbf{(\lambda\ y.\ y\ z)} \rightarrow_\beta$
$(\lambda\ \mathbf{x}.\ \mathbf{x}\ y)\ \mathbf{(\lambda\ y.\ y\ z)} \rightarrow_\beta$
$\mathbf{(\lambda\ y.\ y\ z)}\ y$



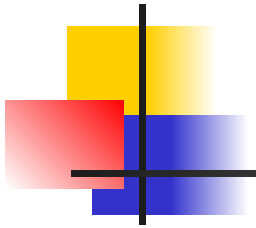
β-reduction

$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms

Example:

$(\lambda\ \mathbf{z}.\ (\lambda\ x.\ x\ y)\ \mathbf{z})\ (\lambda\ \mathbf{y}.\ \mathbf{y\ z}) \rightarrow_\beta$
$(\lambda\ \mathbf{x}.\ \mathbf{x}\ y)\ (\lambda\ \mathbf{y}.\ \mathbf{y\ z}) \rightarrow_\beta$
$(\lambda\ \mathbf{y}.\ \mathbf{y}\ z)\ \mathbf{y} \rightarrow_\beta$
??



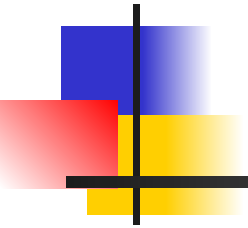
β-reduction

$(\lambda\ x.\ e)\ r \rightarrow_\beta e\ [r / x]$

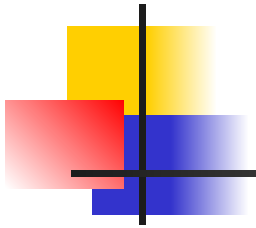
- **Essence of computation** in the lambda calculus
- Usually defined on **α-equivalence classes** of terms

Example:

$(\lambda\ \mathbf{z}.\ (\lambda\ x.\ x\ y)\ \mathbf{z})\ (\lambda\ \mathbf{y}.\ \mathbf{y}\ \mathbf{z}) \rightarrow_\beta$
$(\lambda\ \mathbf{x}.\ \mathbf{x}\ y)\ (\lambda\ \mathbf{y}.\ \mathbf{y}\ \mathbf{z}) \rightarrow_\beta$
$(\lambda\ \mathbf{y}.\ \mathbf{y}\ z)\ \mathbf{y} \rightarrow_\beta$
$\mathbf{y}\ z$

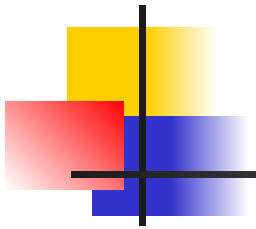


Questions so far?



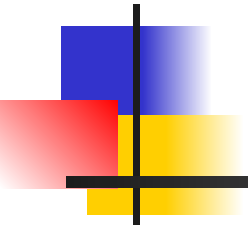
αβ-Equivalence

- **αβ-equivalence** is the smallest congruence containing **α-equivalence** and **β-reduction**
- A term is in **normal form** if no subterm is α-equivalent to a term that can be β-reduced
- Hard fact (**Church-Rosser**): if e_1 and e_2 are αβ-equivalent and both are normal forms, then they are α-equivalent

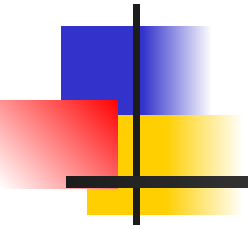


αβ-Equivalence

- **αβ-equivalence** is the smallest congruence containing **α-equivalence** and **β-reduction**
- A term is in **normal form** if no subterm is α-equivalent to a term that can be β-reduced
- Hard fact (**Church-Rosser**): if e_1 and e_2 are αβ-equivalent and both are normal forms, then they are α-equivalent



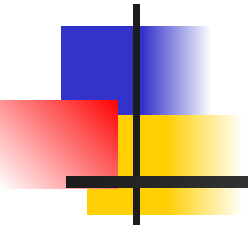
Teaser: Does every term have a normal form?



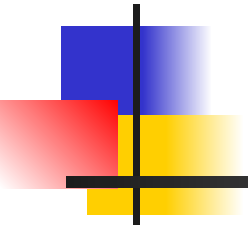
Teaser: Does every term have a normal form?

Try to normalize:

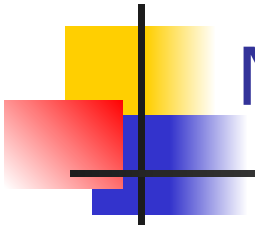
$(\lambda\, x.\ x\ x)\ (\lambda\, x.\ x\ x)$



Questions?



Next Class:
Evaluation in Lambda Calculus
(plus how to write actual programs)



Next Class

- **Please enjoy fall break!**
- **Last quiz** is the Tuesday when you are back.
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help