

# Programming Languages and Compilers (CS 421)

Elsa L Gunter  
2112 SC, UIUC



<https://courses.engr.illinois.edu/cs421/sp2023>

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



# Untyped $\lambda$ -Calculus

---

- How do you compute with the  $\lambda$ -calculus?
- Roughly speaking, by substitution:
- $(\lambda x. e_1) e_2 \Rightarrow^* e_1 [e_2 / x]$
- \* Modulo all kinds of subtleties to avoid free variable capture

# Transition Semantics for $\lambda$ -Calculus

$$\frac{E \rightarrow E''}{EE' \twoheadrightarrow E''E'}$$

- Application (version 1 - Lazy Evaluation)

$$(\lambda x. E) E' \twoheadrightarrow E[E'/x]$$

- Application (version 2 - Eager Evaluation)

$$\frac{E' \twoheadrightarrow E''}{(\lambda x. E) E' \twoheadrightarrow (\lambda x. E) E''}$$

$$\overline{(\lambda x. E) V \twoheadrightarrow E[V/x]}$$

V - variable or abstraction (value)



# How Powerful is the Untyped $\lambda$ -Calculus?

---

- The untyped  $\lambda$ -calculus is Turing Complete
  - Can express any sequential computation
- Problems:
  - How to express basic data: booleans, integers, etc?
  - How to express recursion?
  - Constants, `if_then_else`, etc, are conveniences; can be added as syntactic sugar



# Typed vs Untyped $\lambda$ -Calculus

---

- The *pure*  $\lambda$ -calculus has no notion of type:  $(f f)$  is a legal expression
- Types restrict which applications are valid
- Types are not syntactic sugar! They disallow some terms
- Simply typed  $\lambda$ -calculus is less powerful than the untyped  $\lambda$ -Calculus: NOT Turing Complete (no recursion)



# $\alpha$ Conversion

---

1.  $\alpha$ -conversion:
2.  $\lambda x. \text{exp} \xrightarrow{\alpha} \lambda y. (\text{exp} [y/x])$
3. Provided that
  1.  $y$  is not free in  $\text{exp}$
  2. No free occurrence of  $x$  in  $\text{exp}$  becomes bound in  $\text{exp}$  when replaced by  $y$

$$\lambda x. x (\lambda y. x y) \xrightarrow{\alpha} \lambda y. y (\lambda y. y y)$$

# $\alpha$ Conversion Non-Examples

1. Error:  $y$  is not free in term second

$$\lambda x. x y \not\rightarrow_{\alpha} \lambda y. y y$$

2. Error: free occurrence of  $x$  becomes bound in wrong way when replaced by  $y$

$$\lambda x. \underbrace{\lambda y. x y}_{\text{exp}} \not\rightarrow_{\alpha} \lambda y. \underbrace{\lambda y. y y}_{\text{exp}[y/x]}$$

But  $\lambda x. (\lambda y. y) x \rightarrow_{\alpha} \lambda y. (\lambda y. y) y$

And  $\lambda y. (\lambda y. y) y \rightarrow_{\alpha} \lambda x. (\lambda y. y) x$



# Congruence

---

- Let  $\sim$  be a relation on lambda terms.  $\sim$  is a **congruence** if
- it is an equivalence relation
- If  $e_1 \sim e_2$  then
  - $(e e_1) \sim (e e_2)$  and  $(e_1 e) \sim (e_2 e)$
  - $\lambda x. e_1 \sim \lambda x. e_2$





# $\alpha$ Equivalence

---

- $\alpha$  equivalence is the smallest congruence containing  $\alpha$  conversion
- One usually treats  $\alpha$ -equivalent terms as equal - i.e. use  $\alpha$  equivalence classes of terms



# Example

---

Show:  $\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda y. (\lambda x. x y) y$

■  $\lambda x. (\lambda y. y x) x \dashrightarrow_{\alpha} \lambda z. (\lambda y. y z) z$  so

$\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda z. (\lambda y. y z) z$

■  $(\lambda y. y z) \dashrightarrow_{\alpha} (\lambda x. x z)$  so

$(\lambda y. y z) \sim_{\alpha} (\lambda x. x z)$  so

$(\lambda y. y z) z \sim_{\alpha} (\lambda x. x z) z$  so

$\lambda z. (\lambda y. y z) z \sim_{\alpha} \lambda z. (\lambda x. x z) z$

■  $\lambda z. (\lambda x. x z) z \dashrightarrow_{\alpha} \lambda y. (\lambda x. x y) y$  so

$\lambda z. (\lambda x. x z) z \sim_{\alpha} \lambda y. (\lambda x. x y) y$

■  $\lambda x. (\lambda y. y x) x \sim_{\alpha} \lambda y. (\lambda x. x y) y$



# Substitution

---

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



# Substitution

---

- $x [N / x] = N$
- $y [N / x] = y$  if  $y \neq x$
- $(e_1 e_2) [N / x] = ((e_1 [N / x]) (e_2 [N / x]))$
- $(\lambda x. e) [N / x] = (\lambda x. e)$
- $(\lambda y. e) [N / x] = \lambda y. (e [N / x])$   
provided  $y \neq x$  and  $y$  not free in  $N$ 
  - Rename  $y$  in redex if necessary



# Example

---

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

- Problems?

- z in redex in scope of y binding
- y free in the residue

- $(\lambda y. y z) [(\lambda x. x y) / z] \xrightarrow{\alpha} (\lambda w. w z) [(\lambda x. x y) / z] = \lambda w. w (\lambda x. x y)$



# Example

---

- Only replace free occurrences
- $(\lambda y. y z (\lambda z. z)) [(\lambda x. x) / z] =$   
 $\lambda y. y (\lambda x. x) (\lambda z. z)$

Not

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



# $\beta$ reduction

---

- $\beta$  Rule:  $(\lambda x. P) N \xrightarrow{\beta} P [N / x]$
- Essence of computation in the lambda calculus
- Usually defined on  $\alpha$ -equivalence classes of terms



# Example

---

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

$--\beta--> (\lambda x. x y) (\lambda y. y z)$

$--\beta--> (\lambda y. y z) y --\beta--> y z$

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

$--\beta--> (\lambda x. x x) (\lambda x. x x)$

$--\beta--> (\lambda x. x x) (\lambda x. x x) --\beta--> \dots$





# $\alpha$ $\beta$ Equivalence

---

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



# Order of Evaluation

---

- Not all terms reduce to normal forms
- Not all reduction strategies will produce a normal form if one exists



## Lazy evaluation:

---

- Always reduce the left-most application in a top-most series of applications (i.e. Do not perform reduction inside an abstraction)
- Stop when term is not an application, or left-most application is not an application of an abstraction to a term



# Example 1

---

- $(\lambda z. (\lambda x. x)) ((\lambda y. y y) (\lambda y. y y))$
- Lazy evaluation:
- Reduce the left-most application:
- $(\lambda z. (\lambda x. x)) ((\lambda y. y y) (\lambda y. y y))$   
-- $\beta$ -->  $(\lambda x. x)$



# Eager evaluation

---

- (Eagerly) reduce left of top application to an abstraction
- Then (eagerly) reduce argument
- Then  $\beta$ -reduce the application



# Example 1

---

- $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- Eager evaluation:
- Reduce the rator of the top-most application to an abstraction: Done.
- Reduce the argument:
- $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- $\beta$ -->  $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))$
- $\beta$ -->  $(\lambda z. (\lambda x. x))((\lambda y. y y) (\lambda y. y y))\dots$



## Example 2

---

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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



## Example 2

---

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. \boxed{x} \boxed{x}) \underline{((\lambda y. y y) (\lambda z. z))} \text{ --}\beta\text{-->}$





## Example 2

---

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

$(\lambda x. \boxed{x} \boxed{x}) \underline{((\lambda y. y y) (\lambda z. z))} \text{ --}\beta\text{--}\>$

$\boxed{((\lambda y. y y) (\lambda z. z))} \boxed{((\lambda y. y y) (\lambda z. z))}$



## Example 2

---

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

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



## Example 2

---

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Lazy evaluation:

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

$((\lambda y. \boxed{y} \boxed{y}) \underline{(\lambda z. z)}) ((\lambda y. y y) (\lambda z. z))$



## Example 2

---

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

- Lazy evaluation:

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

$((\lambda y. \boxed{y} \boxed{y}) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\xrightarrow{\beta} (\boxed{(\lambda z. z)} \boxed{(\lambda z. z)})((\lambda y. y y) (\lambda z. z))$



## Example 2

---

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

- Lazy evaluation:

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

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

$\xrightarrow{\beta} ((\lambda z. z) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$



## Example 2

---

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

- Lazy evaluation:

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

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

$\xrightarrow{\beta} ((\lambda z. \boxed{z}) \underline{(\lambda z. z)})((\lambda y. y y) (\lambda z. z))$



## Example 2

---

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

- Lazy evaluation:

$$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{ --}\beta\text{--}\>$$
$$((\lambda y. y y) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$$
$$\text{--}\beta\text{--}\> ((\lambda z. \boxed{z}) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$$
$$\text{--}\beta\text{--}\> \boxed{(\lambda z. z)} ((\lambda y. y y) (\lambda z. z))$$



## Example 2

---

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

- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{ --}\beta\text{--}\rightarrow$

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

$\text{--}\beta\text{--}\rightarrow ((\lambda z. z) (\lambda z. z))((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}\rightarrow (\lambda z. \boxed{z}) ((\lambda y. y y) (\lambda z. z)) \text{ --}\beta\text{--}\rightarrow$

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



## Example 2

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

- Lazy evaluation:

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

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

$\xrightarrow{\beta} ((\lambda z. z) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\xrightarrow{\beta} (\lambda z. z) ((\lambda y. y y) (\lambda z. z)) \xrightarrow{\beta}$

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

## Example 2

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

- Lazy evaluation:

$(\lambda x. x x)((\lambda y. y y) (\lambda z. z)) \text{ --}\beta\text{--}\rightarrow$

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

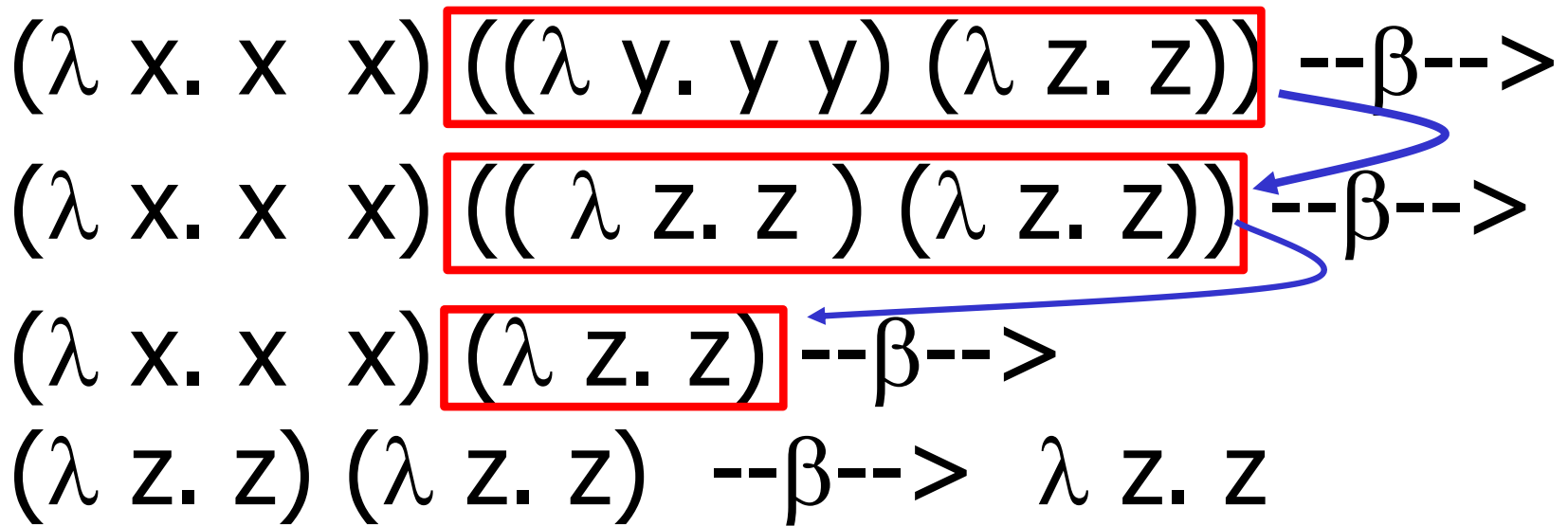
$\text{--}\beta\text{--}\rightarrow ((\lambda z. z) (\lambda z. z)) ((\lambda y. y y) (\lambda z. z))$

$\text{--}\beta\text{--}\rightarrow (\lambda z. z) ((\lambda y. y y) (\lambda z. z)) \text{ --}\beta\text{--}\rightarrow$

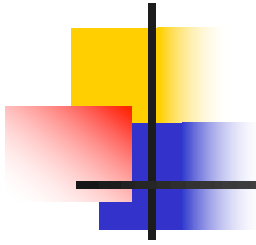
$(\lambda y. y y) (\lambda z. z) \sim\beta\sim \lambda z. z$

# Example 2

- $(\lambda x. x x)((\lambda y. y y) (\lambda z. z))$
- Eager evaluation:



1700 minutes



# Extra Material



# Untyped $\lambda$ -Calculus

---

- Only three kinds of expressions:
  - Variables:  $x, y, z, w, \dots$
  - Abstraction:  $\lambda x. e$   
(Function creation)
  - Application:  $e_1 e_2$



## How to Represent (Free) Data Structures (First Pass - Enumeration Types)

- Suppose  $\tau$  is a type with  $n$  constructors:  
 $C_1, \dots, C_n$  (no arguments)
- Represent each term as an abstraction:
- Let  $C_i \rightarrow \lambda x_1 \dots x_n. x_i$
- Think: you give me what to return in each case (think match statement) and I'll return the case for the  $i$ 'th constructor



# How to Represent Booleans

---

- `bool = True | False`
- `True`  $\rightarrow \lambda x_1. \lambda x_2. x_1 \equiv_{\alpha} \lambda x. \lambda y. x$
- `False`  $\rightarrow \lambda x_1. \lambda x_2. x_2 \equiv_{\alpha} \lambda x. \lambda y. y$

- Notation

- Will write

$\lambda x_1 \dots x_n. e$  for  $\lambda x_1. \dots \lambda x_n. e$

$e_1 e_2 \dots e_n$  for  $(\dots(e_1 e_2) \dots e_n)$





# Functions over Enumeration Types

---

- Write a “match” function

- match  $e$  with  $C_1 \rightarrow x_1$

| ...

|  $C_n \rightarrow x_n$

$\rightarrow \lambda x_1 \dots x_n e. e x_1 \dots x_n$

- Think: give me what to do in each case and give me a case, and I'll apply that case



# Functions over Enumeration Types

- type  $\tau = C_1 | \dots | C_n$
- match  $e$  with  $C_1 \rightarrow x_1$   
| ...  
|  $C_n \rightarrow x_n$
- $match_{\tau} = \lambda x_1 \dots x_n e. e x_1 \dots x_n$
- $e$  = expression (single constructor)  
 $x_i$  is returned if  $e = C_i$



# match for Booleans

---

- `bool = True | False`
- `True`  $\rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x y. x$
- `False`  $\rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x y. y$
  
- `matchbool = ?`



# match for Booleans

---

- `bool = True | False`
- `True`  $\rightarrow \lambda x_1 x_2. x_1 \equiv_{\alpha} \lambda x y. x$
- `False`  $\rightarrow \lambda x_1 x_2. x_2 \equiv_{\alpha} \lambda x y. y$
  
- `matchbool`  $= \lambda x_1 x_2 e. e x_1 x_2$   
 $\equiv_{\alpha} \lambda x y b. b x y$



# How to Write Functions over Booleans

---

- `if b then x1 else x2`  $\rightarrow$
- `if_then_else b x1 x2` = `b x1 x2`
- `if_then_else`  $\equiv \lambda b x_1 x_2 . b x_1 x_2$



# How to Write Functions over Booleans

---

- Alternately:
- if b then  $x_1$  else  $x_2 =$   
match b with True  $\rightarrow x_1$  | False  $\rightarrow x_2 \rightarrow$   
match<sub>bool</sub>  $x_1 x_2 b =$   
 $(\lambda x_1 x_2 b . b x_1 x_2) x_1 x_2 b = b x_1 x_2$
- if\_then\_else  
 $\equiv \lambda b x_1 x_2 . (\text{match}_{\text{bool}} x_1 x_2 b)$   
 $= \lambda b x_1 x_2 . (\lambda x_1 x_2 b . b x_1 x_2) x_1 x_2 b$   
 $= \lambda b x_1 x_2 . b x_1 x_2$



## Example:

---

not b

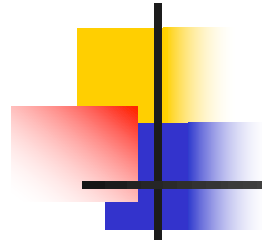
= match b with True -> False | False -> True

→ (match<sub>bool</sub>) False True b

= (λ x<sub>1</sub> x<sub>2</sub> b . b x<sub>1</sub> x<sub>2</sub>) (λ x y. y) (λ x y. x) b

= b (λ x y. y)(λ x y. x)

- not ≡ λ b. b (λ x y. y)(λ x y. x)
- Try and, or



and

or

---





## How to Represent (Free) Data Structures (Second Pass - Union Types)

- Suppose  $\tau$  is a type with  $n$  constructors:  
type  $\tau = C_1 t_{11} \dots t_{1k} \mid \dots \mid C_n t_{n1} \dots t_{nm},$
- Represent each term as an abstraction:
- $C_i t_{i1} \dots t_{ij}, \rightarrow \lambda x_1 \dots x_n. x_i t_{i1} \dots t_{ij},$
- $C_i \rightarrow \lambda t_{i1} \dots t_{ij}, x_1 \dots x_n. x_i t_{i1} \dots t_{ij},$
- Think: you need to give each constructor its arguments first



# How to Represent Pairs

---

- Pair has one constructor (comma) that takes two arguments
- $\text{type } (\alpha, \beta)\text{pair} = (,) \alpha \beta$
- $(a , b) \text{ --> } \lambda x . x a b$
- $(\_ , \_) \text{ --> } \lambda a b x . x a b$



# Functions over Union Types

---

- Write a “match” function
- match  $e$  with  $C_1 y_1 \dots y_{m1} \rightarrow f_1 y_1 \dots y_{m1}$   
| ...  
|  $C_n y_1 \dots y_{mn} \rightarrow f_n y_1 \dots y_{mn}$
- $match\tau \rightarrow \lambda f_1 \dots f_n e. e f_1 \dots f_n$
- Think: give me a function for each case and give me a case, and I'll apply that case to the appropriate function with the data in that case



# Functions over Pairs

---

- $\text{match}_{\text{pair}} = \lambda f p. p f$
- $\text{fst } p = \text{match } p \text{ with } (x,y) \rightarrow x$
- $\text{fst} \rightarrow \lambda p. \text{match}_{\text{pair}} (\lambda x y. x)$   
 $= (\lambda f p. p f) (\lambda x y. x) = \lambda p. p (\lambda x y. x)$
- $\text{snd} \rightarrow \lambda p. p (\lambda x y. y)$



# How to Represent (Free) Data Structures (Third Pass - Recursive Types)

- Suppose  $\tau$  is a type with  $n$  constructors:

type  $\tau = C_1 t_{11} \dots t_{1k} \mid \dots \mid C_n t_{n1} \dots t_{nm},$

- Suppose  $t_{ih} : \tau$  (ie. is recursive)

- In place of a value  $t_{ih}$  have a function to compute the recursive value  $r_{ih} x_1 \dots x_n$

- $C_i t_{i1} \dots r_{ih} \dots t_{ij} \rightarrow \lambda x_1 \dots x_n . x_i t_{i1} \dots (r_{ih} x_1 \dots x_n) \dots t_{ij}$

- $C_i \rightarrow \lambda t_{i1} \dots r_{ih} \dots t_{ij} x_1 \dots x_n . x_i t_{i1} \dots (r_{ih} x_1 \dots x_n) \dots t_{ij},$



# How to Represent Natural Numbers

---

- $\text{nat} = \text{Suc nat} \mid 0$
- $\underline{\text{Suc}} = \lambda n f x. f (n f x)$
- $\text{Suc } n = \lambda f x. f (n f x)$
- $\underline{0} = \lambda f x. x$
- Such representation called *Church Numerals*



# Some Church Numerals

---

- $\overline{\text{Suc } 0} = (\lambda n f x. f (n f x)) (\lambda f x. x) \rightarrow$   
 $\lambda f x. f ((\lambda f x. x) f x) \rightarrow$   
 $\lambda f x. f ((\lambda x. x) x) \rightarrow \lambda f x. f x$

Apply a function to its argument once



# Some Church Numerals

---

■  $\overline{\text{Suc}(\text{Suc } 0)} = (\lambda n f x. f (n f x)) (\text{Suc } 0) \rightarrow$   
 $(\lambda n f x. f (n f x)) (\lambda f x. f x) \rightarrow$   
 $\lambda f x. f ((\lambda f x. f x) f x) \rightarrow$   
 $\lambda f x. f ((\lambda x. f x) x) \rightarrow \lambda f x. f (f x)$

Apply a function twice

In general  $\overline{n} = \lambda f x. f ( \dots (f x) \dots )$  with  $n$  applications of  $f$





# Primitive Recursive Functions

---

- Write a “fold” function
- $\text{fold } f_1 \dots f_n = \text{match } e$   
with  $C_1 y_1 \dots y_{m1} \rightarrow f_1 y_1 \dots y_{m1}$   
| ...  
|  $C_i y_1 \dots r_{ij} \dots y_{in} \rightarrow f_n y_1 \dots (\text{fold } f_1 \dots f_n r_{ij}) \dots y_{mn}$   
| ...  
|  $C_n y_1 \dots y_{mn} \rightarrow f_n y_1 \dots y_{mn}$
- $\text{fold}_\tau \rightarrow \lambda f_1 \dots f_n e. e f_1 \dots f_n$
- Match in non recursive case a degenerate version of fold



# Primitive Recursion over Nat

---

- $\text{fold } f \ z \ n =$
- $\text{match } n \text{ with } 0 \rightarrow z$
- $\quad \quad \quad | \text{Suc } m \rightarrow f (\text{fold } f \ z \ m)$
- $\overline{\text{fold}} \equiv \lambda f \ z \ n. n \ f \ z$
- $\overline{\text{is\_zero}} \ \overline{n} = \overline{\text{fold}} (\lambda r. \overline{\text{False}}) \overline{\text{True}} \ \overline{n}$
- $= (\lambda f \ x. f^n \ x) (\lambda r. \overline{\text{False}}) \overline{\text{True}}$
- $= ((\lambda r. \overline{\text{False}})^n) \overline{\text{True}}$
- $\equiv \text{if } n = 0 \text{ then True else False}$



# Adding Church Numerals

---

- $\bar{n} \equiv \lambda f x. f^n x$     and     $m \equiv \lambda f x. f^m x$

- $\overline{n + m} = \lambda f x. f^{(n+m)} x$   
 $= \lambda f x. f^n (f^m x) = \lambda f x. \bar{n} f (\bar{m} f x)$

- $\bar{+} \equiv \lambda n m f x. n f (m f x)$

- Subtraction is harder



# Multiplying Church Numerals

---

- $\bar{n} \equiv \lambda f x. f^n x$     and     $m \equiv \lambda f x. f^m x$

- $\overline{n * m} = \lambda f x. (f^{n * m}) x = \lambda f x. (f^m)^n x$   
 $= \lambda f x. \bar{n} (\bar{m} f) x$

- $\bar{*} \equiv \lambda n m f x. n (m f) x$



# Predecessor

---

- let pred\_aux n =  
match n with 0 -> (0,0)  
| Suc m  
-> (Suc(fst(pred\_aux m)), fst(pred\_aux m))  
= fold ( $\lambda$  r. (Suc(fst r), fst r)) (0,0) n
- pred  $\equiv$   $\lambda$  n. snd (pred\_aux n) n =  
 $\lambda$  n. snd (fold ( $\lambda$  r.(Suc(fst r), fst r)) (0,0) n)



# Recursion

---

- Want a  $\lambda$ -term  $Y$  such that for all term  $R$  we have
- $Y R = R (Y R)$
- $Y$  needs to have replication to “remember” a copy of  $R$
- $Y = \lambda y. (\lambda x. y(x x)) (\lambda x. y(x x))$
- $Y R = (\lambda x. R(x x)) (\lambda x. R(x x))$   
 $= R ((\lambda x. R(x x)) (\lambda x. R(x x)))$
- Notice: Requires lazy evaluation



# Factorial

---

■ Let  $F = \lambda f n. \text{if } n = 0 \text{ then } 1 \text{ else } n * f (n - 1)$

$Y F 3 = F (Y F) 3$

$= \text{if } 3 = 0 \text{ then } 1 \text{ else } 3 * ((Y F)(3 - 1))$

$= 3 * (Y F) 2 = 3 * (F(Y F) 2)$

$= 3 * (\text{if } 2 = 0 \text{ then } 1 \text{ else } 2 * (Y F)(2 - 1))$

$= 3 * (2 * (Y F)(1)) = 3 * (2 * (F(Y F) 1)) = \dots$

$= 3 * 2 * 1 * (\text{if } 0 = 0 \text{ then } 1 \text{ else } 0 * (Y F)(0 - 1))$

$= 3 * 2 * 1 * 1 = 6$



# Y in OCaml

---

```
# let rec y f = f (y f);;  
val y : ('a -> 'a) -> 'a = <fun>  
# let mk_fact =  
    fun f n -> if n = 0 then 1 else n * f(n-1);;  
val mk_fact : (int -> int) -> int -> int = <fun>  
# y mk_fact;;  
Stack overflow during evaluation (looping  
recursion?).
```





# Eager Eval Y in Ocaml

---

```
# let rec y f x = f (y f) x;;  
val y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b  
= <fun>
```

```
# y mk_fact;;
```

```
- : int -> int = <fun>
```

```
# y mk_fact 5;;
```

```
- : int = 120
```

- Use recursion to get recursion



# Some Other Combinators

---

- For your general exposure
- $I = \lambda x . x$
- $K = \lambda x . \lambda y . x$
- $K_* = \lambda x . \lambda y . y$
- $S = \lambda x . \lambda y . \lambda z . x z (y z)$



---

# End of Extra Material