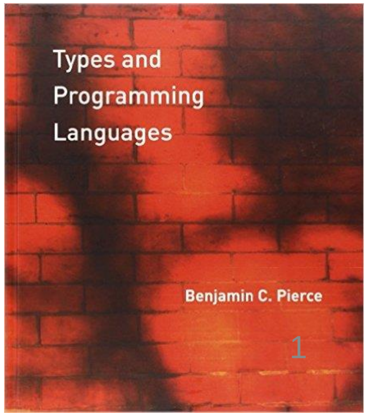
Concepts in Programming Languages – Recitation 6:

Lambda Calculus

Oded Padon & Mooly Sagiv
(original slides by Kathleen Fisher, John Mitchell, Shachar Itzhaky, S. Tanimoto)

Reference:
Types and Programming Languages
by Benjamin C. Pierce, Chapter 5



Untyped Lambda Calculus - Syntax

t ::=	terms
x	variable
λ x. t	abstraction
t t	application

- Terms can be represented as abstract syntax trees
- Syntactic Conventions:
  - Applications associates to left :
    $e_1\ e_2\ e_3 \equiv (e_1\ e_2)\ e_3$
  - The body of abstraction extends as far as possible:
    $\lambda x.\ \lambda y.\ x\ y\ x \equiv \lambda x.\ (\lambda y.\ ((x\ y)\ x))$

Free vs. Bound Variables

- An occurrence of x in t is bound in λx. t
  - otherwise it is free
  - λx is a binder
- FV: t $\rightarrow$ P(Var) is the set free variables of t
  - $FV(x) = \{x\}$
  - $FV(\lambda x.\ t) = FV(t) - \{x\}$
  - $FV(t_1\ t_2) = FV(t_1) \cup FV(t_2)$

Semantics: Substitution, β-reduction, α-conversion

- Substitution

  $[x \mapsto s]\ x = s$

  $[x \mapsto s]\ y = y$ if $y \neq x$

  $[x \mapsto s]\ (\lambda y.\ t_1) = \lambda y.\ [x \mapsto s]\ t_1$ if $y \neq x$ and $y \notin FV(s)$

  $[x \mapsto s]\ (t_1\ t_2) = ([x \mapsto s]\ t_1)\ ([x \mapsto s]\ t_2)$

- β-reduction

  $(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$

- α-conversion

  $(\lambda x.\ t) \Rightarrow_\alpha \lambda y.\ [x \mapsto y]\ t$ if $y \notin FV(t)$

Examples of β-reduction, α-conversion

$(\lambda\ x.\ x)\ y \Rightarrow_{\beta}\ y$

$(\lambda\ x.\ x\ (\lambda\ x.\ x)\ )\ (u\ r) \Rightarrow_{\text{"}\beta+\alpha\text{"}}\ u\ r\ (\lambda\ x.\ x)$

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

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

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

Non-Deterministic Operational Semantics

(E-AppAbs) $(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$ (E-Abs)

(E-App$_1$) $$\frac{t_1 \Rightarrow t'_1}{t_1\ \ t_2 \Rightarrow t'_1\ t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\ \ t_2 \Rightarrow t_1\ t'_2}$$ (E-App$_2$)

Why is this semantics non-deterministic?

Different Evaluation Orders

(E-AppAbs) $(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

$$\frac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'} \quad \text{(E-Abs)}$$

(E-App$_1$) $$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2} \quad \text{(E-App}_2\text{)}$$

$(\lambda x.\ (\text{add } x\ x))\ (\text{add } 2\ 3) \Rightarrow (\lambda x.\ (\text{add } x\ x))\ (5) \Rightarrow \text{add } 5\ 5 \Rightarrow 10$

$(\lambda x.\ (\text{add } x\ x))\ (\text{add } 2\ 3) \Rightarrow (\text{add } (\text{add } 2\ 3)\ (\text{add } 2\ 3)) \Rightarrow$

$(\text{add } 5\ (\text{add } 2\ 3)) \Rightarrow (\text{add } 5\ 5) \Rightarrow 10$

This example: same final result but lazy performs more computations

Different Evaluation Orders

(E-AppAbs) $(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

$$\frac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'}\ \text{(E-Abs)}$$

(E-App$_1$) $$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}\ \text{(E-App}_2\text{)}$$

$(\lambda x.\ \lambda y.\ x)\ 3\ (\text{div}\ 5\ 0) \Rightarrow$ Exception: Division by zero

$(\lambda x.\ \lambda y.\ x)\ 3\ (\text{div}\ 5\ 0) \Rightarrow (\lambda y.\ 3)\ (\text{div}\ 5\ 0) \Rightarrow 3$

This example: lazy suppresses erroneous division and reduces to final result

Can also suppress non-terminating computation.

Many times we want this, for example:

```
if i < len(a) and a[i]==0: print “found zero”
```

Strict	Lazy	Normal Order

Strict

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

precedence

(E-App$_2$)

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

precedence

(E-AppAbs)

$$(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$$

Lazy

(E-AppAbs)

$$(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$$

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

Normal Order

(E-AppAbs)

$$(\lambda\ x.\ t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$$

precedence

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

precedence

(E-App$_2$)

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2}$$

(E-Abs)

$$\frac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'}$$

Call-by-value Operations Semantics via Inductive Definition (no precedence)

t ::= terms

x variable

λ x. t abstraction

t t application

v ::= λ x. t abstraction values

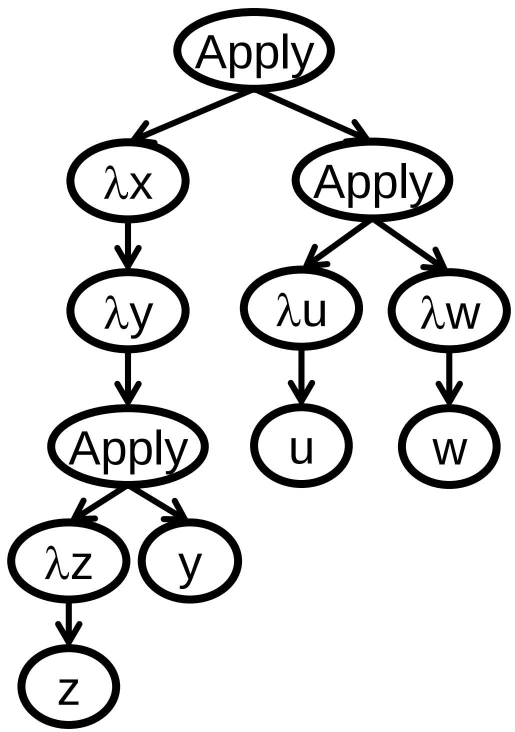
$$(\lambda\ x.\ t_1)\ v_2 \Rightarrow [x \mapsto v_2]\ t_1$$ (E-AppAbs)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ \ t_2\ \Rightarrow\ t'_1\ t_2}$$ (E-APPL1)

$$\frac{t_2 \Rightarrow t'_2}{v_1\ \ t_2\ \Rightarrow\ v_1\ t'_2}$$ (E-APPL2)

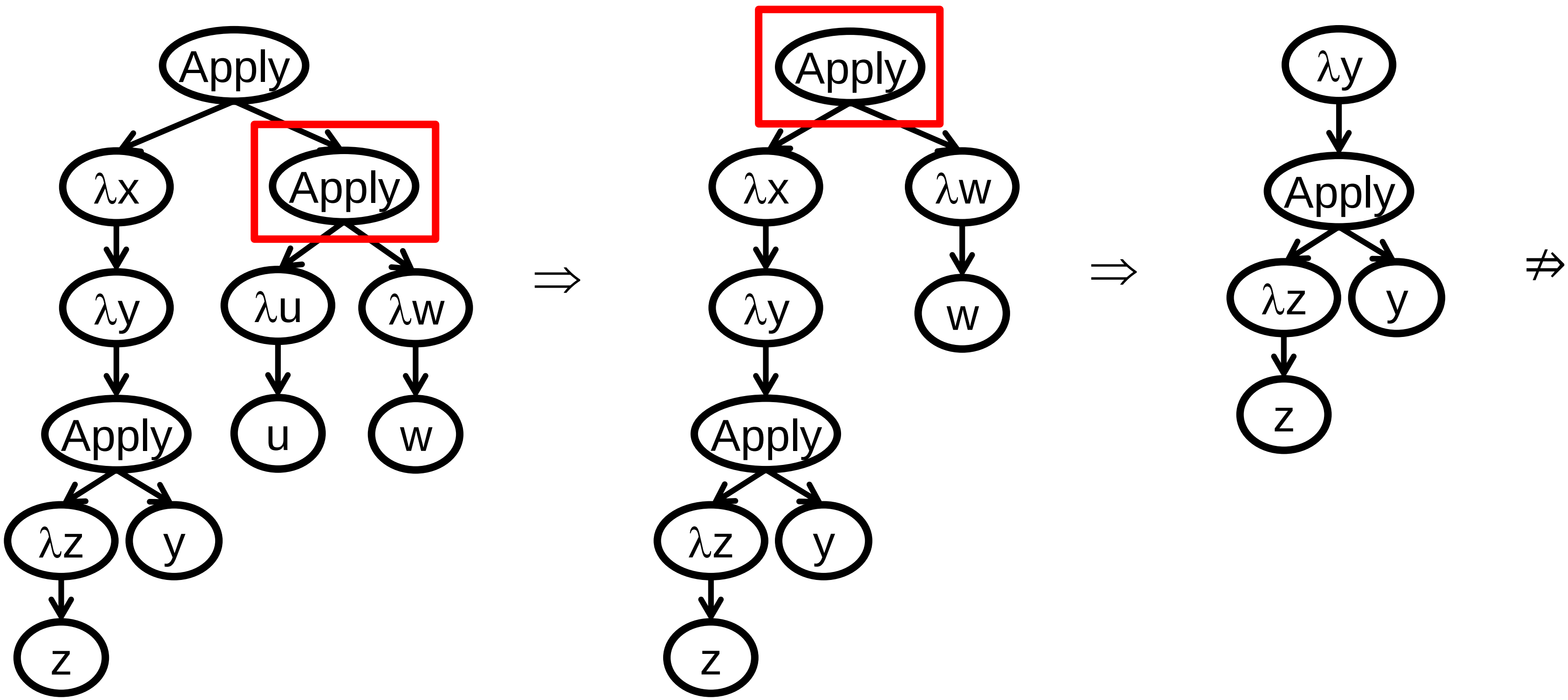
Different Evaluation Orders - Example

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



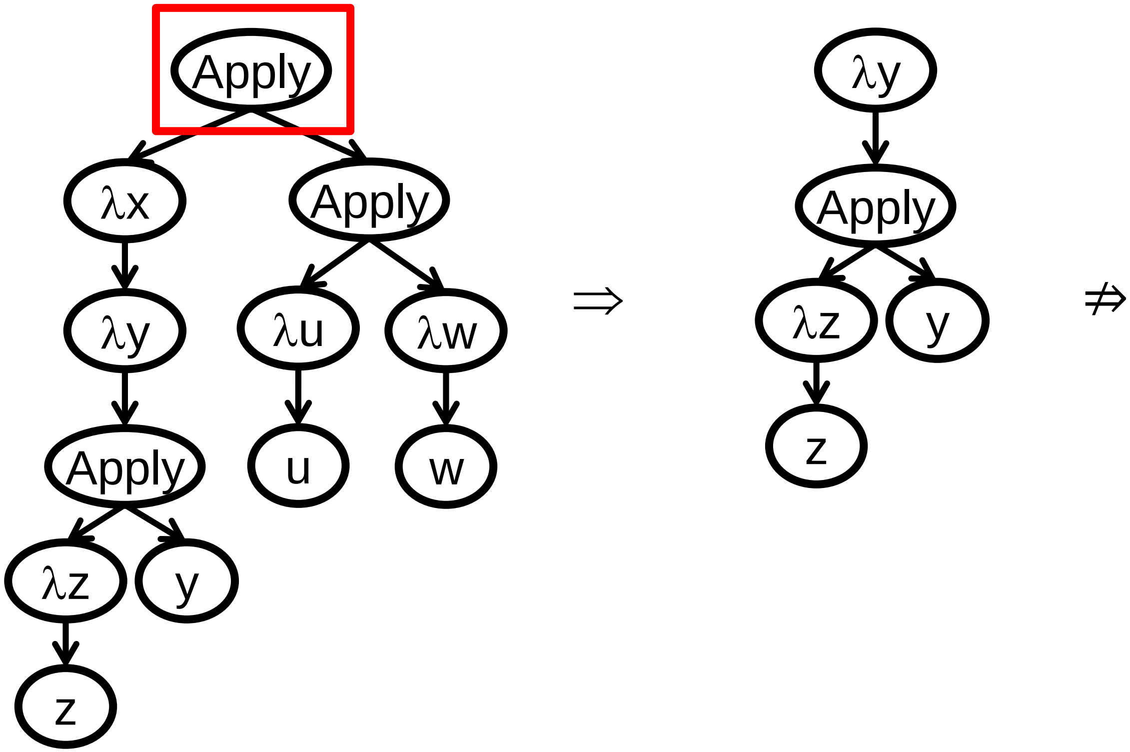
Call By Value

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



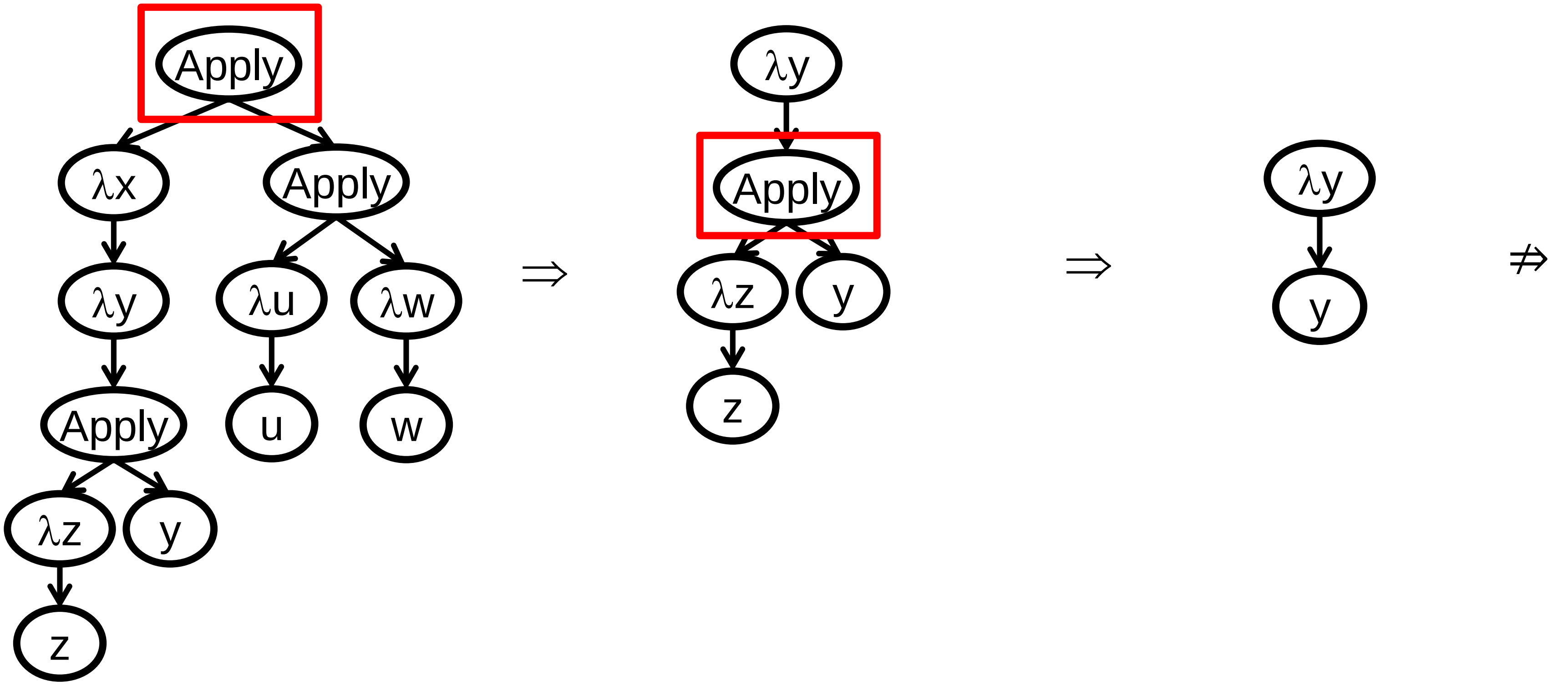
Call By Name (Lazy)

(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Normal Order

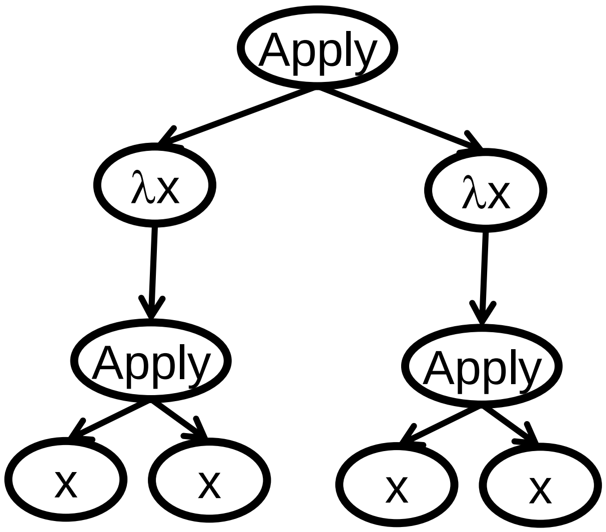
(λ x. λ y. (λ z. z) y) ((λ u. u) (λ w. w))



Divergence

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

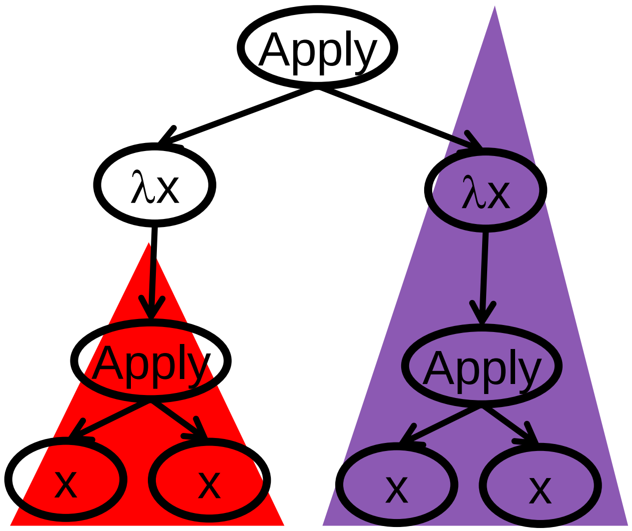
$(\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x))$



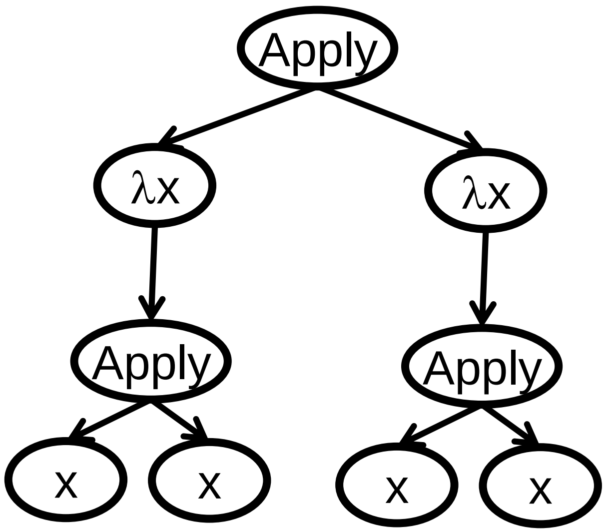
Divergence

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

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



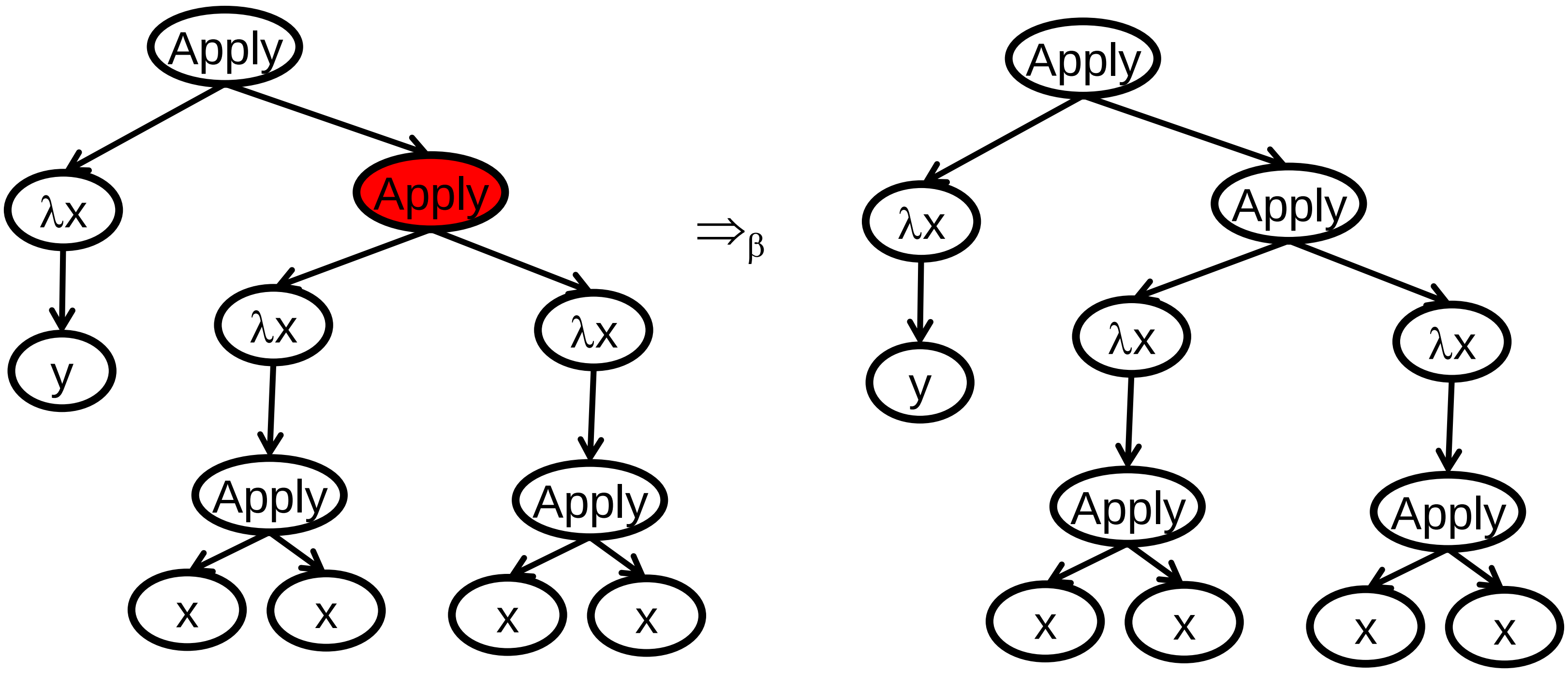
$\Rightarrow_\beta$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

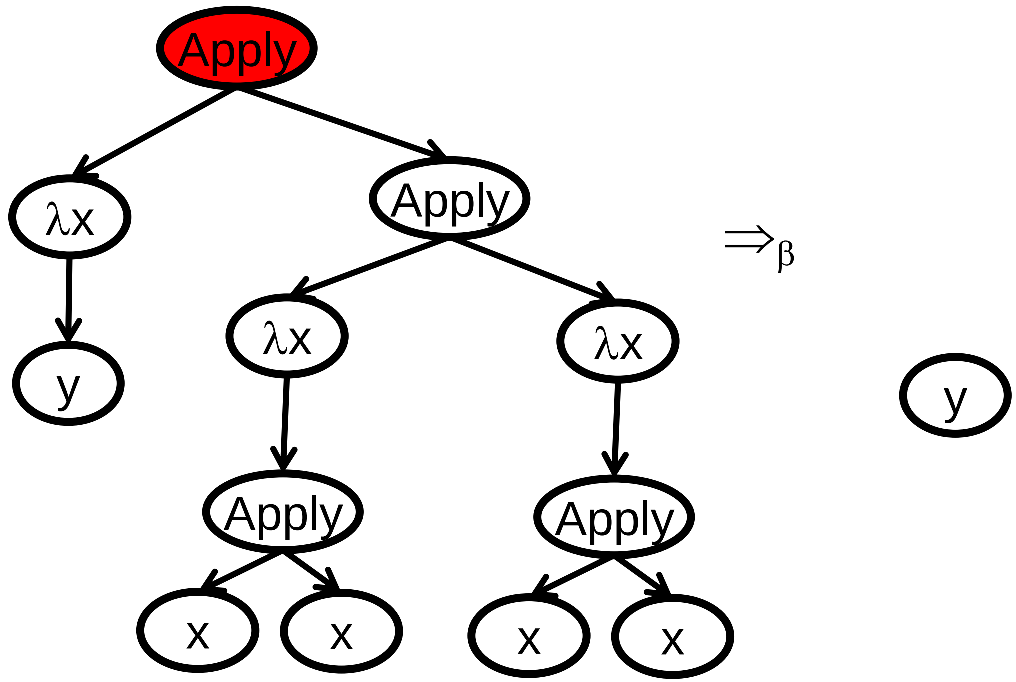
$(\lambda\ x.y)\ ((\lambda\ x.(x\ \ x))\ (\lambda\ x.(x\ \ x)))$



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

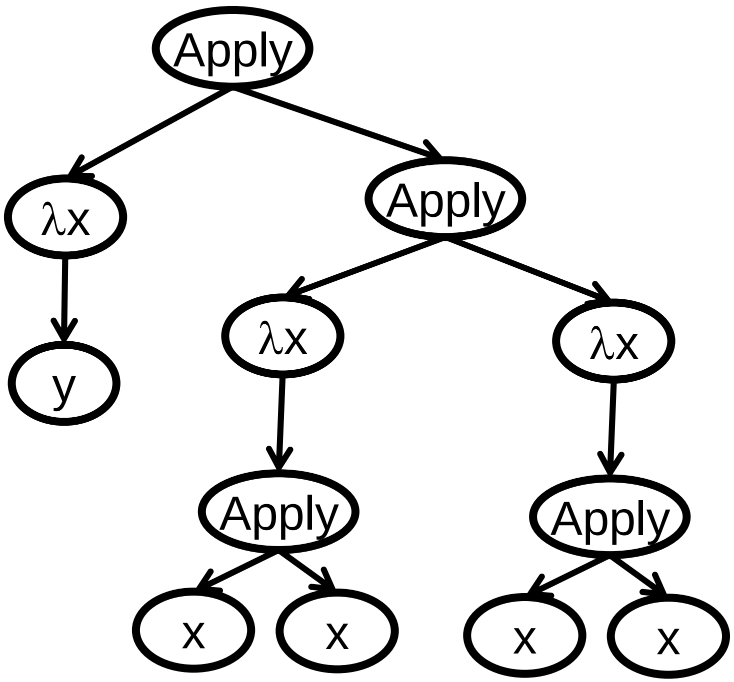
(λ x.y) ((λ x.(x x)) (λ x.(x x)))



Different Evaluation Orders

$(\lambda\ x.\ t_1)\ t_2 \Rightarrow_\beta [x \mapsto t_2]\ t_1$ (β-reduction)

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




```
def f():
    while True: pass

def g(x):
    return 2

print g(f())
```

Summary: Orders of Evaluation

- Full-beta-reduction, Non-deterministic semantics
  - All possible orders
- Call by value, Eager, Strict, Applicative order
  - Left to right
  - Fully evaluate arguments before function
- Normal order
  - The leftmost, outermost redex is always reduced first
- Call by name, Lazy
  - Evaluate arguments as needed
- Call by need
  - Evaluate arguments as needed and store for subsequent usages
  - Implemented in Haskell

Church–Rosser Theorem

If:

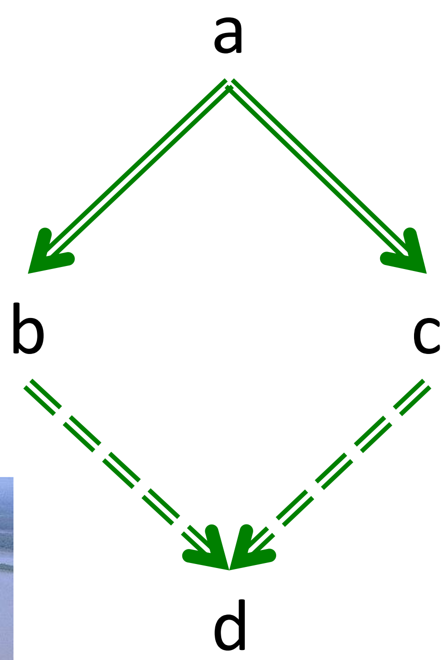
$a \Rightarrow^* b$,

$a \Rightarrow^* c$

then there exists d such that:

$b \Rightarrow^* d$, and

$c \Rightarrow^* d$



Normal Form & Halting Problem

- A term is in normal form if it is stuck in normal order semantics
- Under normal order every term either:
  - Reduces to normal form, or
  - Reduces infinitely
- For a given term, it is undecidable to decide which is the case

Combinators

- A combinator is a function in the Lambda Calculus having no free variables
- Examples
  - $\lambda x.\ x$ is a combinator
  - $\lambda x.\ \lambda y.\ (x\ y)$ is a combinator
  - $\lambda x.\ \lambda y.\ (x\ z)$ is not a combinator
- Combinators can serve nicely as modular building blocks for more complex expressions
- The Church numerals and simulated Booleans are examples of useful combinators

Iteration in Lambda Calculus

- omega = $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$
  - $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \Rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

Y Combinator

Williams + Hirakawa

- Y = $\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$
- Z = $\lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$
- Recursion can be simulated
  - Y only works with call-by-name semantics
  - Z works with call-by-value semantics
- Defining factorial:
  - g = $\lambda f.\ \lambda n.$ if n==0 then 1 else (n * (f (n - 1)))
  - fact = Y g (for call-by-name)
  - fact = Z g (for call-by-value)

Y-Combinator in action (lazy)

Y Combinator

“g = λf. λn. if n==0 then 1 else (n * (f (n - 1)))”

Y = λf. (λx. f (x x)) (λx. f (x x))

Y g v = (λf. (λx. f (x x)) (λx. f (x x))) g v

$\Rightarrow$ ((λx. g (x x)) (λx. g (x x))) v

$\Rightarrow$ (g ((λx. g (x x)) (λx. g (x x)))) v

~ (g (Y g)) v

What happens to Y in strict semantics?

Z-Combinator in action (strict)

"g = λf. λn. if n==0 then 1 else (n * (f (n - 1)))"

Z = λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))

Z g v = (λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))) g v

⇒((λx. g (λy. x x y)) (λx. g (λy. x x y))) v

⇒(g (λy. (λx. g (λy. x x y)) (λx. g (λy. x x y)) y)) v

~ (g (λy. (Z g) y)) v

~ (g (Z g)) v

```
def f1(y):
    return f2(y)
```

Simulating laziness like Z-Combinator

```
def f(x):
    if ask_user("wanna see it?"):
        print x

def g(x, y, z):
    # very expensive computation without side effects

def main():
    # compute a, b, c with side effects
    f(g(a, b, c))
```

- In strict semantics, the above code computes g anyway
  - Lazy will avoid it
- How can achieve this in a strict programming language?

Simulating laziness like Z-Combinator

```
def f(x):
    if ask_user(“?"):
        print x

def g(x, y, z):
    # expensive

def main():
    # compute a, b, c
    f(g(a, b, c))
```

```
def f(x):
    if ask_user("?"):
        print x()

def g(x, y, z):
    # expensive

def main():
    # compute a, b, c
    f(lambda: g(a, b, c))
```

$$Z = \lambda f.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))$$

(E-Abs)

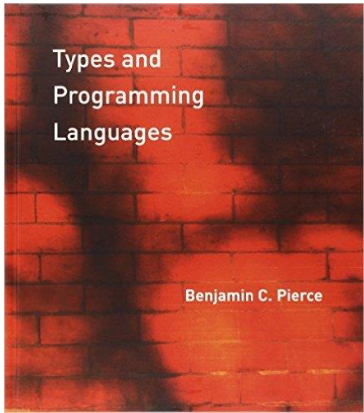
$$\frac{t \Rightarrow t'}{\lambda\ x.\ t \Rightarrow \lambda\ x.\ t'}$$

Typed Lambda Calculus

Chapter 9

Benjamin Pierce

Types and Programming Languages



Simple Types

T ::=		types
	Bool	type of Booleans
	$T \to T$	type of functions

$$T_1 \to T_2 \to T_3 = T_1 \to (T_2 \to T_3)$$

Simple Typed Lambda Calculus

t ::=	terms	T::=	types
x	variable	Bool	atomic type for Booleans
λ x: T. t	abstraction	T $\to$ T	types of functions
t t	application		

Tying Relation $\vdash$

- Type fact: t:T means term t has type T (e.g. 1+3:Int, f:Int$\to$Int)
- Typing relation $\vdash$: relates sets of type facts and type facts
- $\Gamma \vdash$ t:T means under Γ (context, environment), term t has type T
- $\vdash$ t:T means term t has type T under the empty environment (no assumptions)
  - Can t have free variables?

Typing Relation Examples

- Type fact: $t:T$ means term t has type T (e.g. $1+3:Int$, $f:Int \to Int$)
- $\Gamma \vdash t:T$ means under Γ (context, environment), term t has type T
- $\vdash t:T$ means closed term t has type T under the empty environment

Examples

- $x:Int, y:Int \vdash x+y : ?$
- $x:Int, y:Bool \vdash x+y : ?$
- $x:Int \vdash x+y : ?$
- $x:Int, y:Int, b:Bool \vdash$ if b then x else y : ?
- $x:Int, y:Bool, b:Bool \vdash$ if b then x else y : ?
- $x:Int, b:Bool \vdash$ if b then x else y : ?
- $x:Int, f:Int \to Bool \vdash f\ x : ?$
- $x:Int, f: Int \to Int \to Bool \vdash f\ x\ x : ?$
- $x:Int, f: Int \to Int \to Bool \vdash f\ x: ?$
- $f: Int \to Int \to Bool \vdash f\ x\ x : ?$
- $\vdash 1+2 : ?$
- $\vdash$ true : ?
- $x:Int, y:Int, f: Int \to Int \to Bool \vdash 1+2 : ?$

Formally Defining $\vdash$

t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application
T::=	types
$T \rightarrow T$	types of functions
Γ::=	context
$\varnothing$	empty context
Γ, x : T	term variable binding

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\ t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

Adding Booleans

t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application
true	constant true
false	constant false
if t then t else t	conditional

T::=	types
Bool	type of Booleans
$T \to T$	types of functions

Γ::=	context
$\varnothing$	empty context
$\Gamma, x : T$	term variable binding

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.\, t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\, t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\Gamma \vdash \text{true} : \text{Bool} \quad \text{(T-TRUE)}$$

$$\Gamma \vdash \text{false} : \text{Bool} \quad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad \text{(T-IF)}$$

Example Typing Tree

T-VAR $\dfrac{}{f{:}Bool \to Bool,\ x{:}Bool \vdash x{:}Bool}$

T-VAR $\dfrac{}{f{:}Bool \to Bool,\ x{:}Bool \vdash f{:}Bool \to Bool}$

T-FALSE $\dfrac{}{f{:}Bool \to Bool,\ x{:}Bool \vdash false{:}Bool}$

T-VAR $\dfrac{}{f{:}Bool \to Bool,\ x{:}Bool \vdash x{:}Bool}$

T-APP $\dfrac{\bullet \quad \bullet}{f{:}Bool \to Bool,\ x{:}Bool \vdash f\ x : Bool}$

T-IF

$$\dfrac{}{f{:}Bool \to Bool,\ x{:}Bool \vdash \text{if false then } x \text{ else } f\ x : Bool}$$

T-ABS

$$\dfrac{}{f{:}Bool \to Bool \vdash \lambda x{:}Bool\ \ \text{if false then } x \text{ else } f\ x : Bool \to Bool}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \textbf{(T-VAR)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \textbf{(T-APP)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 .\ t_2 : T_1 \to T_2} \textbf{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1{:}Bool \quad \Gamma \vdash t_2{:}T \quad \Gamma \vdash t_3{:}T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \textbf{(T-IF)}$$

$$\Gamma \vdash false : Bool \ \textbf{(T-FALSE)}$$

Curry-Howard Isomorphism

Beautiful connection between type systems (PL) and proof systems (logic)

Propositional Logic	*Simply-Typed Lambda Calculus*
$$\dfrac{\dfrac{\overline{A}^{\,i}}{\vdots}\\ B}{A \rightarrow B}\ (\rightarrow I), i$$	$$\dfrac{\dfrac{\overline{x : A}^{\,i}}{\vdots}\\ B}{\lambda x.M : A \rightarrow B}\ (\rightarrow I), i$$
$$\dfrac{A \quad A \rightarrow B}{B}\ (\rightarrow E)$$	$$\dfrac{N : A \quad M : A \rightarrow B}{MN : B}\ (\rightarrow E)$$

conjunction $A \wedge B$	product type $A \times B$
disjunction $A \vee B$	sum type $A + B$
implication $A \supset B$	function type $A \rightarrow B$

Retrospect

Natural Operational Semantics $\rightarrow$

- $\rightarrow$ is defined **inductively** using **inference rules**, with both **syntactic** conditions on S and **semantic** conditions on s

$$[\text{ass}_{\text{ns}}] \qquad \langle x := a,\ s\rangle \rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$$

$$[\text{skip}_{\text{ns}}] \qquad \langle \texttt{skip},\ s\rangle \rightarrow s$$

$$[\text{comp}_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \rightarrow s',\ \langle S_2,\ s'\rangle \rightarrow s''}{\langle S_1;S_2,\ s\rangle \rightarrow s''}$$

$$[\text{if}^{\text{tt}}_{\text{ns}}] \qquad \frac{\langle S_1,\ s\rangle \rightarrow s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{if}^{\text{ff}}_{\text{ns}}] \qquad \frac{\langle S_2,\ s\rangle \rightarrow s'}{\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \rightarrow s'} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

$$[\text{while}^{\text{tt}}_{\text{ns}}] \qquad \frac{\langle S,\ s\rangle \rightarrow s',\ \langle \texttt{while}\ b\ \texttt{do}\ S,\ s'\rangle \rightarrow s''}{\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \rightarrow s''} \quad \text{if } \mathcal{B}[\![b]\!]s = \mathbf{tt}$$

$$[\text{while}^{\text{ff}}_{\text{ns}}] \qquad \langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \rightarrow s \ \text{ if } \mathcal{B}[\![b]\!]s = \mathbf{ff}$$

Structural Operational Semantics $\Rightarrow$

- $\Rightarrow$ is defined **inductively** using **inference rules**, with both **syntactic** conditions on S and **semantic** conditions on s

$[\text{ass}_{\text{sos}}]$	$\langle x := a,\ s\rangle \Rightarrow s[x \mapsto \mathcal{A}[\![a]\!]s]$
$[\text{skip}_{\text{sos}}]$	$\langle \texttt{skip},\ s\rangle \Rightarrow s$
$[\text{comp}^1_{\text{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow \langle S_1',\ s'\rangle}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_1';S_2,\ s'\rangle}$
$[\text{comp}^2_{\text{sos}}]$	$\dfrac{\langle S_1,\ s\rangle \Rightarrow s'}{\langle S_1;S_2,\ s\rangle \Rightarrow \langle S_2,\ s'\rangle}$
$[\text{if}^{\text{tt}}_{\text{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_1,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{tt}$
$[\text{if}^{\text{ff}}_{\text{sos}}]$	$\langle \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2,\ s\rangle \Rightarrow \langle S_2,\ s\rangle$ if $\mathcal{B}[\![b]\!]s = \mathbf{ff}$
$[\text{while}_{\text{sos}}]$	$\langle \texttt{while}\ b\ \texttt{do}\ S,\ s\rangle \Rightarrow \langle \texttt{if}\ b\ \texttt{then}\ (S;\ \texttt{while}\ b\ \texttt{do}\ S)\ \texttt{else}\ \texttt{skip},\ s\rangle$

λ-Calculus: Non-Deterministic Operational Semantics

(E-AppAbs) $(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

$$\frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'} \quad \text{(E-Abs)}$$

$$\text{(E-App}_1\text{)} \quad \frac{t_1 \Rightarrow t'_1}{t_1\ t_2 \Rightarrow t'_1\ t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{t_1\ t_2 \Rightarrow t_1\ t'_2} \quad \text{(E-App}_2\text{)}$$

λ-Calculus: Lazy Evaluation Operational Semantics

(E-AppAbs) $(\lambda\, x.\, t_1)\ t_2 \Rightarrow [x \mapsto t_2]\ t_1$

(E-App$_1$)
$$\frac{t_1 \Rightarrow t'_1}{t_1\ \ t_2 \Rightarrow t'_1\ t_2}$$

λ-Calculus: Call-by-Value Operational Semantics

(E-AppAbs) $(\lambda\ x.\ t_1)\ v_2 \Rightarrow [x \mapsto v_2]\ t_1$

(E-App$_1$)

$$\frac{t_1 \Rightarrow t'_1}{t_1\ \ t_2 \Rightarrow t'_1\ t_2}$$

$$\frac{t_2 \Rightarrow t'_2}{v_1\ \ t_2 \Rightarrow v_1\ t'_2}$$

(E-App$_2$)

Simply Typed λ-Calculus

t ::=	terms
x	variable
λ x: T. t	abstraction
t t	application
true	constant true
false	constant false
if t then t else t	conditional

T::=	types
Bool	type of Booleans
$T \rightarrow T$	types of functions

Γ::=	context
$\varnothing$	empty context
Γ, x : T	term variable binding

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1 .\ t_2 : T_1 \rightarrow T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

$$\Gamma \vdash \text{true} : \text{Bool} \quad \text{(T-TRUE)}$$

$$\Gamma \vdash \text{false} : \text{Bool} \quad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T} \quad \text{(T-IF)}$$

Retrospect Conclusion:

For the past 7 weeks we have been formalizing programming languages Syntax, Semantics, and Type Rules using ***Inductive Definitions***

Q: What's the difference between a mathematician and a computer scientist?

A: The mathematician sometimes proves theorems without induction ☺