

Fundamentals and lambda calculus



Again: JavaScript functions

- JavaScript functions are first-class
 - Syntax is a bit ugly/terse when you want to use functions as values; recall block scoping:

```
(function () {  
  // ... do something  
})();
```

- New version has cleaner syntax called “arrow functions”
 - Semantics not always the same (`this` has different meaning), but for this class should always be safe to use

In this lecture



In this lecture



In this lecture



In this lecture



What is the lambda calculus?

- Simplest reasonable programming language
 - Only has one feature: functions

Why study it?

- Captures the idea of first-class functions
 - Good system for studying the concept of variable **binding** that appears in almost all languages
- Historically important
 - Competing model of computation introduced by Church as an alternative to Turing machines: substitution (you'll see this today) = symbolic comp
 - Influenced Lisp (thus JS), ML, Haskell, C++, etc.

Why else?

- Base for studying many programming languages
 - You can use lambda calculus and extended it in different ways to study languages and features
 - E.g., we can study the difference between strict languages like JavaScript and lazy ones like Haskell
 - λ + evaluation strategy
 - E.g., we can study different kinds of type systems
 - Simply-typed λ calculus, polymorphic, etc.

Why else?

- Most PL papers describe language models that build on lambda calculus
 - Understanding λ will help you interpret what you are reading in PL research papers
 - Understanding λ will help you get started with other formal/theoretical foundations:
 - Operational semantics
 - Denotational semantics

Before we get started, some terminology

- Syntax (grammar)
- Semantics
- PL implementation: Syntax -> Semantics

Before we get started, some terminology

- Syntax (grammar)
 - The symbols used to write a program
 - E.g., $(x + y)$ is a grammatical expression
- Semantics
- PL implementation: Syntax -> Semantics

Before we get started, some terminology

- Syntax (grammar)
 - The symbols used to write a program
 - E.g., $(x + y)$ is a grammatical expression
- Semantics
 - The actions that occur when a program is executed
- PL implementation: Syntax -> Semantics

Week 2

- Syntax of λ calculus
- Semantics of λ calculus
 - Informal substitution
 - Free and bound variables
 - Formal substitution
 - Evaluation order

Lambda calculus

- Language syntax (grammar):
 - Expressions: $e ::= x \mid \lambda x.e \mid e_1 e_2$
 - Variables: x
 - Functions or λ abstractions: $\lambda x.e$
 - This is the same as $x \Rightarrow e$ in JavaScript!
 - Function application: $e_1 e_2$
 - This is the same as $e_1 (e_2)$ in JavaScript!

Example terms

- ▶ $\lambda x.(2+x)$
 - ▶ Same as: $x \Rightarrow (2 + x)$
- ▶ $(\lambda x.(2 + x)) 5$
 - ▶ Same as: $(x \Rightarrow (2 + x)) (5)$
- ▶ $(\lambda f.(f 3)) (\lambda x.(x + 1))$
 - ▶ Same as: $(f \Rightarrow (f (3))) (x \Rightarrow (x+1))$

Example terms

➤ $\lambda x.(2+x)$

LIES! What is this “2”
and “+”? (Sugar.)

➤ Same as: $x \Rightarrow (2 + x)$

➤ $(\lambda x.(2 + x)) 5$

➤ Same as: $(x \Rightarrow (2 + x)) (5)$

➤ $(\lambda f.(f 3)) (\lambda x.(x + 1))$

➤ Same as: $(f \Rightarrow (f (3))) (x \Rightarrow (x+1))$

Example terms

- ▶ $\lambda x.(2+x)$
 - ▶ Same as: $x \Rightarrow (2 + x)$
- ▶ $(\lambda x.(2 + x)) 5$
 - ▶ Same as: $(x \Rightarrow (2 + x)) (5)$
- ▶ $(\lambda f.(f 3)) (\lambda x.(x + 1))$
 - ▶ Same as: $(f \Rightarrow (f (3))) (x \Rightarrow (x+1))$

Example terms

- $\lambda x.(2+x)$
 - Same as: $x \Rightarrow (2 + x)$
- $(\lambda x.(2 + x)) 5$
 - Same as: $(x \Rightarrow (2 + x)) (5)$
- $(\lambda f.(f 3)) (\lambda x.(x + 1))$
 - Same as: $(f \Rightarrow (f (3))) (x \Rightarrow (x+1))$

Example terms

- $\lambda x.(2+x)$
 - Same as: $x \Rightarrow (2 + x)$
- $(\lambda x.(2 + x)) 5$
 - Same as: $(x \Rightarrow (2 + x)) (5)$
- $(\lambda f.(f 3)) (\lambda x.(x + 1))$
 - Same as: $(f \Rightarrow (f (3))) (x \Rightarrow (x+1))$

JavaScript to λ calculus

- Let's look at function composition: $(f \circ f)(x)$
- In JavaScript:
 - $f \Rightarrow (x \Rightarrow f(f(x)))$
 - $((f \Rightarrow (x \Rightarrow f(f(x))))(x \Rightarrow x+1)) (4)$
- In λ :

JavaScript to λ calculus

- Let's look at function composition: $(f \circ f)(x)$
- In JavaScript:
 - $f \Rightarrow (x \Rightarrow f(f(x)))$
 - $((f \Rightarrow (x \Rightarrow f(f(x))))(x \Rightarrow x+1)) (4)$
- In λ :
 - $\lambda f.(\lambda x. f(f x))$

JavaScript to λ calculus

- Let's look at function composition: $(f \circ f)(x)$
- In JavaScript:
 - $f \Rightarrow (x \Rightarrow f(f(x)))$
 - $((f \Rightarrow (x \Rightarrow f(f(x))))(x \Rightarrow x+1))\ 4$
- In λ :
 - $\lambda f.(\lambda x. f(f x))$
 - $((\lambda f.(\lambda x. f(f x))(\lambda x.x+1))\ 4$

Understanding λ calculus syntax

- λ -calculus syntax: $e ::= x \mid \lambda x. e \mid e_1 e_2$
 - Is $\lambda(x+y). 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. (x x)$ a valid term? (A: yes, B: no)
 - Is $\lambda x. x (x y)$ a valid term? (A: yes, B: no)

Understanding λ calculus syntax

- λ -calculus syntax: $e ::= x \mid \lambda x. e \mid e_1 e_2$
 - Is $\lambda(x+y). 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. (x x)$ a valid term? (A: yes, B: no)
 - Is $\lambda x. x (x y)$ a valid term? (A: yes, B: no)

Understanding λ calculus syntax

- λ -calculus syntax: $e ::= x \mid \lambda x. e \mid e_1 e_2$
 - Is $\lambda(x+y). 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. (x x)$ a valid term? (A: yes, B: no)
 - Is $\lambda x. x (x y)$ a valid term? (A: yes, B: no)

Understanding λ calculus syntax

- λ -calculus syntax: $e ::= x \mid \lambda x. e \mid e_1 e_2$
 - Is $\lambda(x+y). 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. (x x)$ a valid term? (A: yes, B: no)
 - Is $\lambda x. x (x y)$ a valid term? (A: yes, B: no)

Understanding λ calculus syntax

- λ -calculus syntax: $e ::= x \mid \lambda x. e \mid e_1 e_2$
 - Is $\lambda(x+y). 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. 3$ a valid term? (A: yes, B: no)
 - Is $\lambda x. (x x)$ a valid term? (A: yes, B: no)
 - Is $\lambda x. x (x y)$ a valid term? (A: yes, B: no)

More compact syntax

- Function application is left associative
 - $e_1 e_2 e_3 \stackrel{\text{def}}{=} (e_1 e_2) e_3$
- Lambdas binds all the way to right: only stop when you find unmatched closing paren ')'
 - $\lambda x. \lambda y. \lambda z. e \stackrel{\text{def}}{=} \lambda x. (\lambda y. (\lambda z. e))$
 - Why? Lambda abstraction has lowest precedence!

Understanding compact syntax

- Write the parens: $\lambda x.x \ x$

➤ A: $\lambda x.(x \ x)$

➤ B: $(\lambda x.x) \ x$

Understanding compact syntax

- Write the parens: $\lambda x.x \ x$

► A: $\lambda x.(x \ x)$

► B: $(\lambda x.x) \ x$

Understanding compact syntax

- Write the parens: $\lambda y. \lambda x. x \ x =$
 - A: $\lambda y. (\lambda x. x) \ x$
 - B: $\lambda y. (\lambda x. (x \ x))$
 - C: $(\lambda y. (\lambda x. x)) \ x$

Understanding compact syntax

- Write the parens: $\lambda y. \lambda x. x \ x =$

- A: $\lambda y. (\lambda x. x) \ x$

- B: $\lambda y. (\lambda x. (x \ x))$

- C: $(\lambda y. (\lambda x. x)) \ x$

Understanding compact syntax

- Is $(\lambda y. \lambda x. x) x = \lambda y. \lambda x. x x$?

- A: yes
- B: no

Understanding compact syntax

- Is $(\lambda y. \lambda x. x) x = \lambda y. \lambda x. x x$?

➤ A: yes

➤ B: no

Parsing rules

- Applications are left associative
- Precedence: application > lambda abstraction

Add parentheses

- $\lambda y. \lambda x. x \times \lambda z. y \lambda w. w$
- $\lambda y. \lambda x. (x \times) \lambda z. y \lambda w. w$
- $\lambda y. \lambda x. ((x \times) \lambda z. y) \lambda w. w$
- $\lambda y. \lambda x. (((x \times) \lambda z. y) \lambda w. w)$
- A: $\lambda y. \lambda x. (((x \times) (\lambda z. y)) (\lambda w. w))$
- B: $\lambda y. \lambda x. (((x \times) (\lambda z. y) (\lambda w. w)))$

Add parentheses

- $\lambda y. \lambda x. x \times \lambda z. y \lambda w. w$
- $\lambda y. \lambda x. (x \times) \lambda z. y \lambda w. w$
- $\lambda y. \lambda x. ((x \times) \lambda z. y) \lambda w. w$
- $\lambda y. \lambda x. (((x \times) \lambda z. y) \lambda w. w)$
- $\lambda y. \lambda x. (((x \times) \lambda z. y) (\lambda w. w))$
- A: $\lambda y. \lambda x. (((x \times) (\lambda z. y)) (\lambda w. w))$
- B: $\lambda y. \lambda x. (((x \times) (\lambda z. y) (\lambda w. w)))$

Add parentheses

- $\lambda y. \lambda x. x \times \lambda z. y \lambda w. w$
- $\lambda y. \lambda x. (x \times) \lambda z. y \lambda w. w$
- $\lambda y. \lambda x. ((x \times) \lambda z. y) \lambda w. w$
- $\lambda y. \lambda x. (((x \times) \lambda z. y) \lambda w. w)$
- $\lambda y. \lambda x. (((x \times) \lambda z. y) (\lambda w. w))$
- A: $\lambda y. \lambda x. (((x \times) (\lambda z. y)) (\lambda w. w))$
- B: $\lambda y. \lambda x. (((x \times) (\lambda z. y) (\lambda w. w)))$

↑ First unmatched closing paren

Even more compact syntax

- Can always variables left of the period
 - $\lambda x.\lambda y.\lambda z.e \stackrel{\text{def}}{=} \lambda xyz.e$
- This makes the term look like a 3 argument function
 - Can implement multiple-argument function using single-argument functions: called currying (bonus)
- We won't use this syntax, but you may see in the wild

Week 2

- Syntax of λ calculus ✓
- Semantics of λ calculus
 - Informal substitution
 - Free and bound variables
 - Formal substitution
 - Evaluation order

Semantics of λ calculus

- Reduce a term to another as much as we can
 - If we can't reduce it any further, the term is said to be in **normal form**
- How? Rewrite terms!
 - What does that mean?! Substitution!

Example terms

- Example: $(\lambda x.(2 + x)) \textcolor{brown}{5}$
 - In JavaScript: `(x => (2 + x)) (5)`
- Example: $(\lambda f.(f \ 3)) \ (\lambda x.(x + 1))$
 - In JavaScript: `(f => (f (3))) (\textcolor{brown}{x} => (x+1))`

Example terms

- Example: $(\lambda x.(2 + x)) \ 5$
 - In JavaScript: $(x \Rightarrow (2 + x)) \ (5) \rightarrow (2 + 5)$
- Example: $(\lambda f.(f \ 3)) \ (\lambda x.(x + 1))$
 - In JavaScript: $(f \Rightarrow (f \ (3))) \ (x \Rightarrow (x+1))$

Example terms

- Example: $(\lambda x.(2 + x)) \ 5$
 - In JavaScript: $(x \Rightarrow (2 + x)) \ (5) \rightarrow (2 + 5) \rightarrow 7$
- Example: $(\lambda f.(f \ 3)) \ (\lambda x.(x + 1))$
 - In JavaScript: $(f \Rightarrow (f \ (3))) \ (\lambda x.(x + 1))$

Example terms

- Example: $(\lambda x.(2 + x)) \ 5 \rightarrow (2 + 5)$
 - In JavaScript: `(x => (2 + x))(5) → (2 + 5) → 7`
- Example: $(\lambda f.(f\ 3))\ (\lambda x.(x + 1))$
 - In JavaScript: `(f => (f(3)))(x => (x+1))`

Example terms

- Example: $(\lambda x.(2 + x)) \ 5 \rightarrow (2 + 5) \rightarrow 7$
 - In JavaScript: `(x => (2 + x))(5) → (2 + 5) → 7`
- Example: $(\lambda f.(f\ 3))\ (\lambda x.(x + 1))$
 - In JavaScript: `(f => (f(3)))(x => (x+1))`

Example terms

- Example: $(\lambda x.(2 + x)) \ 5 \rightarrow (2 + 5) \rightarrow 7$
 - In JavaScript: `(x => (2 + x)) (5) → (2 + 5) → 7`
- Example: $(\lambda f.(f\ 3))\ (\lambda x.(x + 1))$
 - In JavaScript: `(f => (f (3))) (\x => (x+1))`
 $\rightarrow ((\textcolor{brown}{x} \Rightarrow (x+1)) \ (\textcolor{blue}{3}))$

Example terms

- Example: $(\lambda x.(2 + x)) \ 5 \rightarrow (2 + 5) \rightarrow 7$
 - In JavaScript: $(x \Rightarrow (2 + x)) \ (5) \rightarrow (2 + 5) \rightarrow 7$
- Example: $(\lambda f.(f \ 3)) \ (\lambda x.(x + 1))$
 - In JavaScript:
$$\begin{aligned} & (f \Rightarrow (f \ (3))) \ (x \Rightarrow (x+1)) \\ & \quad \rightarrow ((x \Rightarrow (x+1)) \ (3)) \\ & \quad \rightarrow (3+1) \rightarrow 4 \end{aligned}$$

Example terms

- Example: $(\lambda x.(2 + x)) \ 5 \rightarrow (2 + 5) \rightarrow 7$
 - In JavaScript: $(x \Rightarrow (2 + x)) \ (5) \rightarrow (2 + 5) \rightarrow 7$
- Example: $(\lambda f.(f \ 3)) \ (\lambda x.(x + 1))$
 - $\rightarrow ((\lambda x.(x + 1)) \ 3)$
- Example: $(\lambda f.(f \ 3)) \ (\lambda x.(x + 1))$
 - In JavaScript:
 - $f \Rightarrow (f \ (3))$
 - $x \Rightarrow (x+1)$
 - $\rightarrow ((x \Rightarrow (x+1)) \ (3))$
 - $\rightarrow (3+1) \rightarrow 4$

Example terms

- Example: $(\lambda x.(2 + x)) \textcolor{brown}{5} \rightarrow (2 + \textcolor{brown}{5}) \rightarrow 7$
 - In JavaScript: $(x \Rightarrow (2 + x)) (\textcolor{brown}{5}) \rightarrow (2 + \textcolor{brown}{5}) \rightarrow 7$
- Example: $(\lambda f.(f \ 3)) (\lambda x.(x + 1))$
 - $\rightarrow ((\lambda x.(x + 1)) \ 3)$
 - $\rightarrow (\textcolor{blue}{3} + 1) \rightarrow 4$
- In JavaScript:
 - $f \Rightarrow (f \ (3)) \ (x \Rightarrow (x+1))$
 - $\rightarrow ((\textcolor{red}{x} \Rightarrow (x+1)) \ (\textcolor{blue}{3}))$
 - $\rightarrow (\textcolor{blue}{3+1}) \rightarrow 4$

Easy! Pattern for function application:
substitute the term you are applying the
function to for the argument variable

Substitution (not right)

- Def: Substitution: $e_1 [x := e_2]$
 - Replace every occurrence of x in e_1 with e_2
- General reduction rule for λ calculus:
 - $(\lambda x.e_1) e_2 \rightarrow$
 - Function application rewritten to e_1 (the function body) with every x in e_1 substituted with e_2 (argument)

Substitution (not right)

- Def: Substitution: $e_1 [x := e_2]$
 - Replace every occurrence of x in e_1 with e_2
- General reduction rule for λ calculus:
 - $(\lambda x.e_1) e_2 \rightarrow e_1 [x := e_2]$
 - Function application rewritten to e_1 (the function body) with every x in e_1 substituted with e_2 (argument)

Simple examples

- Reduce $(\lambda x.(2 + x)) 5$
- Reduce: $(\lambda x.\lambda y.\lambda z.y+3) 4 5 6 =$

Simple examples

- Reduce $(\lambda x.(2 + x)) 5 \rightarrow (2 + 5) \rightarrow 7$
- Reduce: $(\lambda x.\lambda y.\lambda z.y+3) 4 5 6 =$

Simple examples

- Reduce $(\lambda x.(2 + x)) 5 \rightarrow (2 + 5) \rightarrow 7$
- Reduce: $(\lambda x.\lambda y.\lambda z.y+3) 4 5 6 =$
 $((((\lambda x.\lambda y.\lambda z.y+3) 4) 5) 6 \rightarrow$
 $((\lambda y.\lambda z.y+3) 5) 6 \rightarrow$ $(\lambda z.5+3) 6 \rightarrow (5+3) \rightarrow 8$

Simple examples (cont)

- Reduce $(\lambda x.(\lambda y.2) 3) 5$
- Reduce: $((\lambda x.(\lambda y.2)) 3) 5$
- Is $(\lambda x.(\lambda y.2) 3) 5 = ((\lambda x.(\lambda y.2)) 3) 5$?
 - A: yes, B: no

Simple examples (cont)

- Reduce $(\lambda x.(\lambda y.2) 3) 5 \rightarrow (\lambda x. 2) 5 \rightarrow 2$
- Reduce: $((\lambda x.(\lambda y.2)) 3) 5$
- Is $(\lambda x.(\lambda y.2) 3) 5 = ((\lambda x.(\lambda y.2)) 3) 5$?
 - A: yes, B: no

Simple examples (cont)

- Reduce $(\lambda x.(\lambda y.2) 3) 5 \rightarrow (\lambda x. 2) 5 \rightarrow 2$
- Reduce: $((\lambda x.(\lambda y.2)) 3) 5 \rightarrow ((\lambda y.2) 5) \rightarrow 2$
- Is $(\lambda x.(\lambda y.2) 3) 5 = ((\lambda x.(\lambda y.2)) 3) 5$?
 - A: yes, B: no

Simple examples (cont)

- Reduce $(\lambda x.(\lambda y.2) 3) 5 \rightarrow (\lambda x. 2) 5 \rightarrow 2$
- Reduce: $((\lambda x.(\lambda y.2)) 3) 5 \rightarrow ((\lambda y.2) 5) \rightarrow 2$
- Is $(\lambda x.(\lambda y.2) 3) 5 = ((\lambda x.(\lambda y.2)) 3) 5$?
 - A: yes, B: no

A more complicated example

A more complicated example

- Reduce the following:

➤ $(\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{4}$

A more complicated example

- Reduce the following:

➤ $(\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{4}$

➤ $(\lambda a. \textcolor{blue}{4} + a) 7$

A more complicated example

- Reduce the following:

➤ $(\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{4}$

➤ $(\lambda a. \textcolor{blue}{4} + a) 7$

➤ $\textcolor{blue}{4} + 7$

A more complicated example

- Reduce the following:

- $(\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{4}$
- $(\lambda a. \textcolor{blue}{4} + a) 7$
- $\textcolor{blue}{4} + 7$
- 11

Let's make this even more fun!

Let's make this even more fun!

- Instead of 4, let's apply function to $(a + 5)$
 - $(\lambda x. (\lambda a. x + a) 7) \text{ (a + 5)}$

Let's make this even more fun!

- Instead of 4, let's apply function to $(a + 5)$
 - $(\lambda x. (\lambda a. x + a) 7) \text{ (a + 5)}$
 - $(\lambda a. \text{ (a + 5)} + a) 7$

Let's make this even more fun!

- Instead of 4, let's apply function to $(a + 5)$
 - $(\lambda x. (\lambda a. x + a) 7) (a + 5)$
 - $(\lambda a. (a + 5) + a) 7$
 - $(7 + 5) + 7$

Let's make this even more fun!

- Instead of 4, let's apply function to $(a + 5)$
 - $(\lambda x. (\lambda a. x + a) 7) \text{ (a + 5)}$
 - $(\lambda a. \text{ (a + 5)} + a) 7$
 - $\text{ (7 + 5)} + 7$
 - 19

Let's make this even more fun!

- Instead of 4, let's apply function to $(a + 5)$

➤ $(\lambda x. (\lambda a. x + a) 7) \text{ (a + 5)}$

➤ $(\lambda a. \text{ (a + 5)} + a) 7$

➤ $\text{ (7 + 5)} + 7$

Is this right?

➤ 19

A: yes, B: no

Let's make this even more fun!

- Instead of 4, let's apply function to $(a + 5)$
 - $(\lambda x. (\lambda a. x + a) 7) \text{ (a + 5)}$
 - $(\lambda a. \text{ (a + 5)} + a) 7$
 - $\text{ (7 + 5)} + 7$
 - 19

Is this right?

A: yes, B: no

Substitution is surprisingly complex

- Recall our reduction rule for application:
 - $(\lambda x. e_1) e_2 \rightarrow e_1 [x := e_2]$
 - This function application reduces to e_1 (the function body) where every x in e_1 is substituted with e_2 (value we're applying func to)
 - Where did we go wrong? When we substituted:
 - $(\lambda x. (\lambda a. x + a) 7) (a + 5)$
 - $(\lambda a. (a + 5) + a) 7$ the a is **captured!**

Example to do at home

Example to do at home

- Reduce the following
 - $((\lambda f.(\lambda x. f (f x))) (\lambda x.x+1)) 4$

Example to do at home

- Reduce the following
 - $((\lambda f.(\lambda x. f (f x))) (\lambda x.x+1)) 4$
 - $(\lambda x. (\lambda x.x+1) ((\lambda x.x+1) x)) 4$

Example to do at home

- Reduce the following
 - $((\lambda f.(\lambda x. f (f x))) (\lambda x.x+1)) 4$
 - $(\lambda x. (\lambda x.x+1) ((\lambda x.x+1) x)) 4$
 - $(\lambda x.x+1) ((\lambda x.x+1) 4)$

Example to do at home

- Reduce the following
 - $((\lambda f.(\lambda x. f (f x))) (\lambda x.x+1)) 4$
 - $(\lambda x. (\lambda x.x+1) ((\lambda x.x+1) x)) 4$
 - $(\lambda x.x+1) ((\lambda x.x+1) 4)$
 - $(\lambda x.x+1) (4+1)$

Example to do at home

- Reduce the following

- $((\lambda f. (\lambda x. f (f x))) (\lambda x. x+1)) \ 4$
- $(\lambda x. (\lambda x. x+1) ((\lambda x. x+1) x)) \ 4$
- $(\lambda x. x+1) ((\lambda x. x+1) \ 4)$
- $(\lambda x. x+1) (4+1)$
- $4+1+1$

Example to do at home

- Reduce the following

- $((\lambda f.(\lambda x. f (f x))) (\lambda x.x+1)) 4$
- $(\lambda x. (\lambda x.x+1) ((\lambda x.x+1) x)) 4$
- $(\lambda x.x+1) ((\lambda x.x+1) 4)$
- $(\lambda x.x+1) (4+1)$
- $4+1+1$
- 6

Example to do at home

Example to do at home

- Reduce the following
 - $(\lambda f.(\lambda x. f (f x))) (\lambda y.y+x)$

Example to do at home

- Reduce the following
 - $(\lambda f.(\lambda x. f (f x))) (\lambda y.y+x)$
 - $\lambda x. (\lambda y.y+x) ((\lambda y.y+x) x)$

Example to do at home

- Reduce the following
 - $(\lambda f.(\lambda x. f (f x))) (\lambda y.y+x)$
 - $\lambda x. (\lambda y.y+x) ((\lambda y.y+x) x)$
 - $\lambda x. (\lambda y.y+x) (x+x)$

Example to do at home

- Reduce the following
 - $(\lambda f.(\lambda x. f (f x))) (\lambda y.y+x)$
 - $\lambda x. (\lambda y.y+x) ((\lambda y.y+x) x)$
 - $\lambda x. (\lambda y.y+x) (x+x)$
 - $\lambda x. (x+x+x)$

Example to do at home

- Reduce the following
 - $(\lambda f.(\lambda x. f (f x))) (\lambda y.y+x)$
 - $\lambda x. (\lambda y.y+x) ((\lambda y.y+x) x)$
 - $\lambda x. (\lambda y.y+x) (x+x)$
 - $\lambda x. (x+x+x)$

????
that's not a function that adds
x to argument two times

Another way to see the problem

- Syntactic sugar: $\text{let } x = e_1 \text{ in } e_2 \stackrel{\text{def}}{=} (\lambda x. e_2) e_1$
- Let syntax makes this easy to see:

➤ $\begin{array}{l} \text{let } x = a + 5 \text{ in} \\ \quad \text{let } \textcolor{red}{a} = 7 \text{ in} \\ \quad \quad x + a \end{array}$ \rightarrow $\begin{array}{l} \text{let } x = a + 5 \text{ in} \\ \quad \text{let } \textcolor{red}{a} = 7 \text{ in} \\ \quad \quad (a + 5) + \textcolor{red}{a} \end{array}$

Another way to see the problem

- Syntactic sugar: $\text{let } x = e_1 \text{ in } e_2 \stackrel{\text{def}}{=} (\lambda x. e_2) e_1$
- Let syntax makes this easy to see:
 - $\begin{array}{l} \text{let } x = a + 5 \text{ in} \\ \quad \text{let } \textcolor{red}{a} = 7 \text{ in} \\ \quad \quad x + a \end{array} \rightarrow \begin{array}{l} \text{let } x = a + 5 \text{ in} \\ \quad \text{let } \textcolor{red}{a} = 7 \text{ in} \\ \quad \quad (a + 5) + \textcolor{red}{a} \end{array}$
 - Very obviously wrong!
 - But, guess what: your C macro preprocessor does this!

Fixing the problem

- How can we fix this?

1. Rename variables!

➤ let $x = a+5$ in
let $a = 7$ in
 $x + a$

2. Do the “dumb” substitution!

Fixing the problem

- How can we fix this?

1. Rename variables!

➤ let $x = a+5$ in let $x = a+5$ in
let $a = 7$ in \rightarrow let $a123 = 7$ in
 $x + a$ $x + a123$

2. Do the “dumb” substitution!

Fixing the problem

- How can we fix this?

1. Rename variables!

► let $x = a+5$ in let $x = a+5$ in let $x = a+5$ in
let $a = 7$ in \rightarrow let $a123 = 7$ in \rightarrow let $a123 = 7$ in
 $x + a$ $x + a123$ $(a+5) + a123$

2. Do the “dumb” substitution!

Why is this the way to go?

- Def: variable x is bound in $\lambda x.(x+y)$
 - Can we always rename bound variables? A: yes, B: no
- Yes! Bound variables are just “placeholders”
 - Above: x is not special, we could have used z
 - We say they are equivalent: $\lambda x.(x+y) =_{\alpha} \lambda z.(z+y)$
- Renaming amounts to converting bound variable names to avoid capture: e.g., $\lambda x.(x+y)$ to $\lambda z.(z+y)$

Can we rename everything?

- Can we rename y in $\lambda x.(x+y)$? (A: yes, B: no)



- Intuition:



Can we rename everything?

- Can we rename y in $\lambda x.(x+y)$? (A: yes, B: no)
 - No! We don't know what y may be, so we must keep it as is!
- Intuition:
 -
 -

Can we rename everything?

- Can we rename y in $\lambda x.(x+y)$? (A: yes, B: no)
 - No! We don't know what y may be, so we must keep it as is!
- Intuition:
 - Can change the name of your function argument variables but not of variables from the outer scope
 - E.g., $\forall \textcolor{blue}{x}. P(\textcolor{blue}{x}, \textcolor{red}{y})$ or $\sum_{i \in \{1, \dots, 10\}} \textcolor{red}{x}_i + \textcolor{blue}{y}$

Week 2

- Syntax of λ calculus ✓
- Semantics of λ calculus
 - Informal substitution ✓
 - Free and bound variables
 - Formal substitution
 - Evaluation order

Let's think about this more formally

Def: free variables

- If a variable is not bound by a λ , we say that it is **free**
 - e.g., y is free in $\lambda x.(x+y)$
 - is x free? A: yes, B: no
- We can compute the free variables of any term:
 - $FV(x) = \{x\}$
 - $FV(\lambda x.e) = FV(e) \setminus \{x\}$ think: build out!
 - $FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$

Def: free variables

- If a variable is not bound by a λ , we say that it is **free**
 - e.g., y is free in $\lambda x.(x+y)$
 - is x free? A: yes, B: no
- We can compute the free variables of any term:
 - $FV(x) = \{x\}$
 - $FV(\lambda x.e) = FV(e) \setminus \{x\}$ think: build out!
 - $FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$

Def: free variables

- If a variable is not bound by a λ , we say that it is **free**
 - e.g., y is free in $\lambda x.(x+y)$
 - is x free? A: yes, B: no
- We can compute the free variables of any term:
 - $FV(x) = \{x\}$
 - $FV(\lambda x.e) = FV(e) \setminus \{x\}$ think: build out!
 - $FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$

Def: free variables

- If a variable is not bound by a λ , we say that it is **free**
 - e.g., y is free in $\lambda x.(x+y)$
 - is x free? No! We say x is bound in $\lambda x.(x+y)$
- We can compute the free variables of any term:
 - $FV(x) = \{x\}$
 - $FV(\lambda x.e) = FV(e) \setminus \{x\}$ think: build out!
 - $FV(e_1 e_2) = FV(e_1) \cup FV(e_2)$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:

- $x[x := e] =$
- $y[x := e] =$
- $(e_1 \ e_2)[x := e] =$
- $(\lambda x. e_1)[x := e] =$
- $(\lambda y. e_1)[x := e_2] =$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:

- $x[x := e] = e$
- $y[x := e] =$
- $(e_1 \ e_2)[x := e] =$
- $(\lambda x. e_1)[x := e] =$
- $(\lambda y. e_1)[x := e_2] =$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:

- $x[x := e] = e$
- $y[x := e] = y$ if $y \neq x$
- $(e_1 e_2)[x := e] =$
- $(\lambda x. e_1)[x := e] =$
- $(\lambda y. e_1)[x := e_2] =$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
 - $x[x := e] = e$
 - $y[x := e] = y$ if $y \neq x$
 - $(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$
 - $(\lambda x. e_1)[x := e] =$
 - $(\lambda y. e_1)[x := e_2] =$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
 - $x[x := e] = e$
 - $y[x := e] = y$ if $y \neq x$
 - $(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$
 - $(\lambda x. e_1)[x := e] = \lambda x. e_1$
 - $(\lambda y. e_1)[x := e_2] =$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
 - $x[x := e] = e$
 - $y[x := e] = y$ if $y \neq x$
 - $(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$
 - $(\lambda x. e_1)[x := e] = \lambda x. e_1$
 - $(\lambda y. e_1)[x := e_2] = \lambda y. e_1[x := e_2]$ if $y \neq x$ and $y \notin FV(e_2)$

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
 - $x[x := e] = e$
 - $y[x := e] = y$ if $y \neq x$
 - $(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$
 - $(\lambda x. e_1)[x := e] = \lambda x. e_1$
 - $(\lambda y. e_1)[x := e_2] = \lambda y. e_1[x := e_2]$ if $y \neq x$ and $y \notin FV(e_2)$
 - Why the if?

Def: Capture-avoiding substitution

- Capture-avoiding substitution:
 - $x[x := e] = e$
 - $y[x := e] = y$ if $y \neq x$
 - $(e_1 e_2)[x := e] = (e_1[x := e]) (e_2[x := e])$
 - $(\lambda x. e_1)[x := e] = \lambda x. e_1$
 - $(\lambda y. e_1)[x := e_2] = \lambda y. e_1[x := e_2]$ if $y \neq x$ and $y \notin FV(e_2)$
 - Why the if? If y is free in e_2 this would capture it!

Lambda calculus: equational theory

- α -renaming or α -conversion
 - $\lambda x.e = \lambda y.e[x:=y]$ where $y \notin FV(e)$
- β -reduction
 - $(\lambda x.e_1) e_2 = e_1[x:=e_2]$
- η -conversion
 - $\lambda x.(e x) = e$ where $x \notin FV(e)$
- We define our \rightarrow relation using these equations!

Back to our example

Back to our example

- What we should have done:
 - $(\lambda x. (\lambda a. x + a) 7) \text{ (a + 5)}$

Back to our example

- What we should have done:

$$\triangleright (\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\alpha} (\lambda x. (\lambda b. x + b) 7) \textcolor{blue}{(a + 5)}$$

Back to our example

- What we should have done:

$$\triangleright (\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\alpha} (\lambda x. (\lambda b. x + b) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\beta} (\lambda b. \textcolor{blue}{(a + 5)} + b) 7$$

Back to our example

- What we should have done:

$$\triangleright (\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\alpha} (\lambda x. (\lambda b. x + b) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\beta} (\lambda b. \textcolor{blue}{(a + 5)} + b) 7$$

$$=_{\beta} \textcolor{blue}{(a + 5)} + 7$$

Back to our example

- What we should have done:

$$\triangleright (\lambda x. (\lambda a. x + a) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\alpha} (\lambda x. (\lambda b. x + b) 7) \textcolor{blue}{(a + 5)}$$

$$=_{\beta} (\lambda b. \textcolor{blue}{(a + 5)} + b) 7$$

$$=_{\beta} \textcolor{blue}{(a + 5)} + 7$$

$$=_{\beta} a + 12$$

The to do at home example

The to do at home example

- What you should have done:
 - $(\lambda f.(\lambda x. f (f x))) (\lambda y.y+x)$

The to do at home example

- What you should have done:

➤ $(\lambda f. (\lambda x. f (f x))) (\lambda y. y + x)$

$=\alpha (\lambda f. (\lambda z. f (f z))) (\lambda y. y + x)$

The to do at home example

- What you should have done:

$$\triangleright (\lambda f. (\lambda x. f (f x)) \text{ (}\lambda y. y + x\text{)}$$

$$=\alpha (\lambda f. (\lambda z. f (f z)) \text{ (}\lambda y. y + x\text{)}$$

$$=\beta \lambda z. \text{ (}\lambda y. y + x\text{) } ((\lambda y. y + x) z)$$

The to do at home example

- What you should have done:

$$\triangleright (\lambda f. (\lambda x. f (f x)) \text{ (}\lambda y. y + x\text{) }$$

$$=_{\alpha} (\lambda f. (\lambda z. f (f z)) \text{ (}\lambda y. y + x\text{) }$$

$$=_{\beta} \lambda z. \text{ (}\lambda y. y + x\text{) } ((\lambda y. y + x) z)$$

$$=_{\beta} \lambda z. \text{ (}\lambda y. y + x\text{) } (z + x)$$

The to do at home example

- What you should have done:

$$\triangleright (\lambda f. (\lambda x. f (f x))) (\lambda y. y + x)$$

$$=\alpha (\lambda f. (\lambda z. f (f z))) (\lambda y. y + x)$$

$$=\beta \lambda z. (\lambda y. y + x) ((\lambda y. y + x) z)$$

$$=\beta \lambda z. (\lambda y. y + x) (z + x)$$

$$=\beta \lambda z. z + x + x$$

Week 2

- Syntax of λ calculus ✓
- Semantics of λ calculus
 - Informal substitution ✓
 - Free and bound variables ✓
 - Formal substitution ✓
 - Evaluation order

Evaluation order

- What should we reduce first in $(\lambda x.x) ((\lambda y.y) z)$?
 - A: The inner term: $(\lambda y.y) z$
 - B: The outer term: $(\lambda x.x) ((\lambda y.y) z)$

Evaluation order

- What should we reduce first in $(\lambda x.x) ((\lambda y.y) z)$?
 - A: The inner term: $(\lambda y.y) z$
 - B: The outer term: $(\lambda x.x) ((\lambda y.y) z)$
- Does it matter?
 - No! They both reduce to z !

- Church-Rosser Theorem: “If you reduce to a **normal form**, it doesn’t matter what order you do the reductions.” This is known as confluence.
- Does this mean that reduction order doesn’t matter for any program?
 - A: yes B: no

- Church-Rosser Theorem: “If you reduce to a **normal form**, it doesn’t matter what order you do the reductions.” This is known as confluence.
- Does this mean that reduction order doesn’t matter for any program?
 - A: yes B: no

Does evaluation order really not matter?

Does evaluation order really not matter?

- Consider a curious term called Ω
 - $\Omega \stackrel{\text{def}}{=} (\lambda x. x\ x) \ (\color{blue}{\lambda x. x\ x})$

Does evaluation order really not matter?

- Consider a curious term called Ω

➤ $\Omega \stackrel{\text{def}}{=} (\lambda x. x\ x) \ (\color{blue}{\lambda x. x\ x})$

$$=\beta (x\ x)[\ x:= \color{blue}{(\lambda x. x\ x)}]$$

Does evaluation order really not matter?

- Consider a curious term called Ω

➤ $\Omega \stackrel{\text{def}}{=} (\lambda x. x\ x) \ (\lambda x. x\ x)$

$$= \beta \ (x\ x)[\ x := (\lambda x. x\ x)]$$

$$= \beta \ (\lambda x. x\ x) \ (\lambda x. x\ x)$$

Does evaluation order really not matter?

- Consider a curious term called Ω

➤ $\Omega \stackrel{\text{def}}{=} (\lambda x. x\ x) \ (\lambda x. x\ x)$

$$= \beta \ (x\ x)[\ x := (\lambda x. x\ x)]$$

$$= \beta \ (\lambda x. x\ x) \ (\lambda x. x\ x)$$

$$= \Omega$$

Deja vu!

$$\Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega$$

(Ω has no normal form)

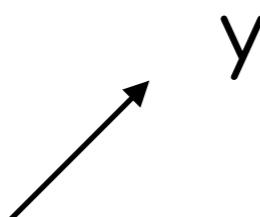
Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω ?

$(\lambda x.y) \Omega$

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω ?

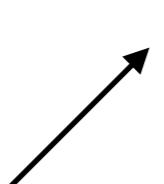
$$(\lambda x.y) \Omega$$


A diagram illustrating the application of a function. A horizontal line segment points upwards and to the right, ending in an arrowhead. Above the arrowhead is the variable y . Below the arrowhead is the expression $(\lambda x.y) \Omega$, representing the application of the function $(\lambda x.y)$ to the argument Ω .

Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω ?

$$(\lambda x.y) \Omega \xrightarrow{\hspace{1cm}} (\lambda x.y) \textcolor{red}{\Omega}$$

 y

Does evaluation order really not matter?

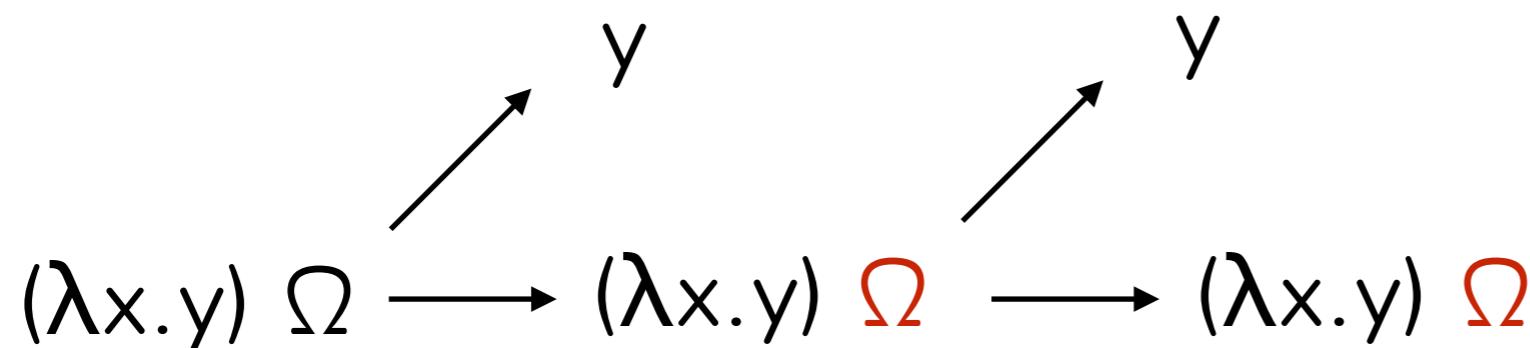
- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω ?

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

The diagram shows the expression $(\lambda x.y) \Omega$ followed by a horizontal arrow pointing to $(\lambda x.y) \Omega$. Above the first Ω , there is a black arrow pointing upwards to the variable y . Above the second Ω , there is a red arrow pointing upwards to the same variable y .

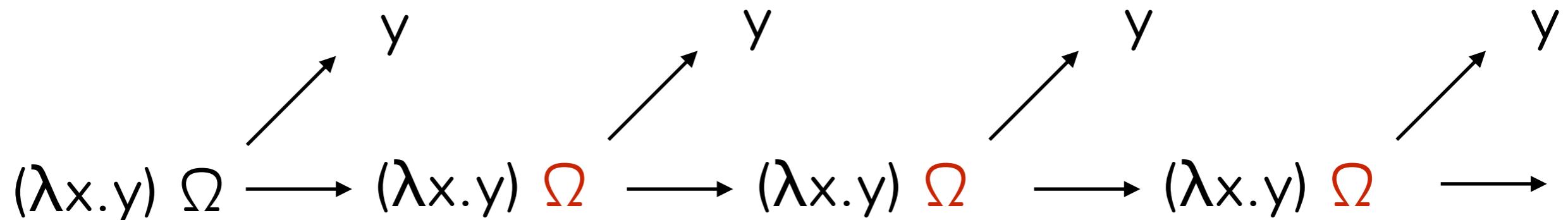
Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω ?



Does evaluation order really not matter?

- Consider a function that ignores its argument: $(\lambda x.y)$
- What happens when we call it on Ω ?



Does evaluation order really not matter?

- Nope! Evaluation order does matter!

Call-by-value

- Reduce function, then reduce args, then apply
 - $e_1 \ e_2$
- JavaScript's evaluation strategy is call-by-value (ish)
 - What does this program do?
 - $(x \Rightarrow 33) \ ((x \Rightarrow x(x)) \ (x \Rightarrow x(x)))$

Call-by-value

- Reduce function, then reduce args, then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x. e_1') \ e_2$
- JavaScript's evaluation strategy is call-by-value (ish)
 - What does this program do?
 - $(x \Rightarrow 33) \ ((x \Rightarrow x(x)) \ (x \Rightarrow x(x)))$

Call-by-value

- Reduce function, then reduce args, then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ n$
- JavaScript's evaluation strategy is call-by-value (ish)
 - What does this program do?
 - $(x \Rightarrow 33) \ ((x \Rightarrow x(x)) \ (x \Rightarrow x(x)))$

Call-by-value

- Reduce function, then reduce args, then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ n \rightarrow e_1'[x:=n]$
- JavaScript's evaluation strategy is call-by-value (ish)
 - What does this program do?
 - $(x \Rightarrow 33) \ ((x \Rightarrow x(x)) \ (x \Rightarrow x(x)))$

Call-by-value

- Reduce function, then reduce args, then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ n \rightarrow e_1'[x:=n]$
- JavaScript's evaluation strategy is call-by-value (ish)
 - What does this program do?
 - $(x \Rightarrow 33) \ ((x \Rightarrow x(x)) \ (x \Rightarrow x(x)))$
 - RangeError: Maximum call stack size exceeded

Call-by-name

- Reduce function then apply
 - $e_1\ e_2$
- Haskell's evaluation strategy is call-by-name
 - It only does what is absolutely necessary!
 - Actually it's call-by-need = call-by-name + sharing

Call-by-name

- Reduce function then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x. e_1') \ e_2$
- Haskell's evaluation strategy is call-by-name
 - It only does what is absolutely necessary!
 - Actually it's call-by-need = call-by-name + sharing

Call-by-name

- Reduce function then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x.e_1') \ e_2 \rightarrow e_1'[x:=e_2]$
- Haskell's evaluation strategy is call-by-name
 - It only does what is absolutely necessary!
 - Actually it's call-by-need = call-by-name + sharing

Call-by-name

- Reduce function then apply
 - $e_1 \ e_2 \rightarrow \dots \rightarrow (\lambda x.e'_1) \ e_2 \rightarrow e'_1[x:=e_2] \rightarrow \dots$
- Haskell's evaluation strategy is call-by-name
 - It only does what is absolutely necessary!
 - Actually it's call-by-need = call-by-name + sharing

Summary

- A term may have many **redexes** (subterms can reduce)
 - Evaluation strategy says which redex to evaluate
 - Evaluation not guaranteed to find normal form
- Call-by-value: evaluate function & args before β reduce
- Call-by-name: evaluate function, then β -reduce

Today

- Syntax of λ calculus ✓
- Semantics of λ calculus ✓
 - Free and bound variables ✓
 - Substitution ✓
 - Evaluation order ✓

Takeaway

- λ -calculus is a formal system
 - “Simplest reasonable programming language” - Ramsey
 - Binders show up everywhere!
 - Know your capture-avoiding substitution!

Bonus!

Recursive functions in λ -calculus

- Suppose you want to implement factorial in λ -calculus
 - $\lambda n. \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{fac } (n-1))$
 - Is this right? A: yes, B: no

Recursive functions in λ -calculus

- Suppose you want to implement factorial in λ -calculus
 - $\lambda n. \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{fac } (n-1))$
 - Is this right? A: yes, B: no
 - Why not? fac is not a thing!

Recursive functions in λ -calculus

- Suppose you want to implement factorial in λ -calculus
 - $\lambda n. \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{fac } (n-1))$
 - Is this right? A: yes, B: no
 - Why not? fac is not a thing!

The Y combinator

About

Y Combinator created a new model for funding early stage startups.

Twice a year we invest a small amount of money (**\$120k**) in a large number of startups.

The startups move to Silicon Valley for 3 months, during which we work intensively with them to get the company into the best possible shape and refine their pitch to investors. This cycle culminates in Demo Day, where the startups present their companies to a carefully selected, invite-only audience.

But YC doesn't end on Demo Day. We and the YC alumni network continue to help founders for the life of their company, and beyond.



Y combinator

- Like Ω , there is a special term Y in lambda calculus
 - $Y \stackrel{\text{def}}{=} \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$
- Why is this interesting? It reduces as follows
 - $Y f$
 $=\beta f(Yf)$
 - $=\beta f(f(Yf))$
 - $=\beta f(\dots f(Yf)\dots)$

Y combinator

- Like Ω , there is a special term Y in lambda calculus
 - $Y \stackrel{\text{def}}{=} \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$
- Why is this interesting? It reduces as follows
 - $$\begin{aligned} & Y f \\ & =_{\beta} f(Yf) \\ & =_{\beta} f(f(Yf)) \\ & =_{\beta} f(\dots f(Yf)\dots) \end{aligned}$$

Def: Fixed-point of a function f is a value x such that $f x = x$

How can we use this?

- Let's go back to factorial example.

➤ $f \stackrel{\text{def}}{=} \lambda \text{fac}. \lambda n. \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{fac} (n-1))$

➤ $\text{factorial} \stackrel{\text{def}}{=} Y f$

$=\beta f (Y f)$

$= f \text{ factorial}$

$= (\lambda \text{fac}. \lambda n. \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{fac} (n-1))) \text{ factorial}$

$=\beta \lambda n. \text{if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{factorial} (n-1))$

How can we use this?

- Apply as usual:

➤ factorial 2

= β (f (Y f)) 2

= β ($\lambda n.$ if $n \leq 1$ then 1 else $n * (\text{factorial } (n-1))$) 2

= β if $2 \leq 1$ then 1 else $2 * (\text{factorial } (2-1))$

= β if $2 \leq 1$ then 1 else $2 * (\text{factorial } 1)$

= β $2 * (\text{factorial } 1)$

How can we use this?

- Apply as usual:

$$= \beta\ 2 * (\text{factorial } 1)$$
$$= \beta\ 2 * ((f \text{ factorial}) 1)$$
$$= \beta\ 2 * ((\lambda n. \text{ if } n \leq 1 \text{ then } 1 \text{ else } n * (\text{factorial } (n-1))) 1)$$
$$= \beta\ 2 * (\text{if } 1 \leq 1 \text{ then } 1 \text{ else } 1 * (\text{factorial } (1-1)))$$
$$= \beta\ 2 * (\text{if } 1 \leq 1 \text{ then } 1 \text{ else } 1 * (\text{factorial } 0))$$
$$= \beta\ 2 * 1$$
$$= \beta\ 2$$