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 | λx.e | e_1 e_2
    - Variables: x
    - Functions or λ abstractions: λx.e
      - This is the same as x => e in JavaScript!
    - Function application: e_1 e_2
      - This is the same as e_1 (e_2) in JavaScript!

Example terms

- λx.(2+x)
  - Same as: x => (2 + x)
- (λx.(2 + x)) 5
  - Same as: (x => (2 + x)) (5)
- (λf.(f 3)) (λx.(x + 1))
  - Same as: (f => (f (3))) (x => (x+1))

Example terms

- λx.(2+x)

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

  - Same as: `x => (2 + x)`

- (λx.(2 + x)) 5
  - Same as: `(x => (2 + x)) (5)`

- (λf.(f 3)) (λx.(x + 1))
  - Same as: `(f => (f (3))) (x => (x+1))`

Example terms

- λx.(2+x)
  - Same as: x => (2 + x)
- (λx.(2 + x)) 5
  - Same as: (x => (2 + x)) (5)
- (λf.(f 3)) (λx.(x + 1))
  - Same as: (f => (f (3))) (x => (x+1))

Example terms

- λx.(2+x)
  - Same as: x => (2 + x)
- (λx.(2 + x)) 5
  - Same as: (x => (2 + x)) (5)
- (λf.(f 3)) (λx.(x + 1))
  - Same as: (f => (f (3))) (x => (x+1))

Example terms

- λx.(2+x)
  - Same as: `x => (2 + x)`
- (λx.(2 + x)) 5
  - Same as: `(x => (2 + x)) (5)`
- (λf.(f 3)) (λx.(x + 1))
  - Same as: `(f => (f (3))) (x => (x+1))`

JavaScript to λ calculus

- Let's look at function composition: `(f ∘ f)(x)`
- In JavaScript:
  - `f => (x => f (f (x)))`
  - `((f => (x => f (f (x)))) (x => x+1)) (4)`
- In λ:

JavaScript to λ calculus

- Let's look at function composition: `(f ∘ f)(x)`
- In JavaScript:
  - `f => (x => f (f (x)))`
  - `((f => (x => f (f (x)))) (x => x+1)) (4)`
- In λ:
  - λf.(λx. f (f x))

JavaScript to λ calculus

- Let's look at function composition: `(f ∘ f)(x)`
- In JavaScript:
  - `f => (x => f (f (x)))`
  - `((f => (x => f (f (x)))) (x => x+1)) (4)`
- In λ:
  - λf.(λx. f (f x))
  - ((λf.(λx. f (f x)) (λx.x+1)) 4

Understanding λ calculus syntax

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

Understanding λ calculus syntax

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

Understanding λ calculus syntax

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

Understanding λ calculus syntax

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

Understanding λ calculus syntax

- λ-calculus syntax: e ::= x | λx.e | e_1 e_2
  - Is λ(x+y). 3 a valid term? (A: yes, B: no)
  - Is λx. 3 a valid term? (A: yes, B: no)
  - Is λx. (x x) a valid term? (A: yes, B: no)
  - Is λx. 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 ‘)’
  - $\lambda x.\lambda y.\lambda z.e \stackrel{\text{def}}{=} \lambda x.(\lambda y.(\lambda z.e))$

Understanding compact syntax

- Write the parens: λx.x x
  - A: λx.(x x)
  - B: (λx.x) x

Understanding compact syntax

- Write the parens: λx.x x
  - A: λx.(x x)
  - B: (λx.x) x

Understanding compact syntax

- Write the parens: λy.λx.x x =
  - A: λy.(λx.x) x
  - B: λy.(λx.(x x))
  - C: (λy.(λx.x)) x

Understanding compact syntax

- Write the parens: λy.λx.x x =
  - A: λy.(λx.x) x
  - B: λy.(λx.(x x))
  - C: (λy.(λx.x)) x

Understanding compact syntax

- Is (λy.λx.x) x = λy.λx.x x ?
  - A: yes
  - B: no

Understanding compact syntax

- Is (λy.λx.x) x = λy.λx.x x ?
  - A: yes
  - B: no

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: (λx.(2 + x)) 5
  - In JavaScript: `(x => (2 + x)) (5)`
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: `(f => (f (3))) (x => (x+1))`

Example terms

- Example: (λx.(2 + x)) 5
  - In JavaScript: `(x => (2 + x)) (5)` → `(2 + 5)`
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: `(f => (f (3))) (x => (x+1))`

Example terms

- Example: (λx.(2 + x)) 5
  - In JavaScript: `(x => (2 + x)) (5)` → `(2 + 5)` → `7`
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: `(f => (f (3))) (x => (x+1))`

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5)
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)
    → (3+1) → 4

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  → ((λx.(x + 1)) 3)
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)
    → (3+1) → 4

Example terms

- Example: (λx.(2 + x)) 5 → (2 + 5) → 7
  - In JavaScript: (x => (2 + x)) (5) → (2 + 5) → 7
- Example: (λf.(f 3)) (λx.(x + 1))
  → ((λx.(x + 1)) 3)
  → (3 + 1) → 4
  - In JavaScript: (f => (f (3))) (x => (x+1))
    → ((x => (x+1)) (3)
    → (3+1) → 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 (λx.(2 + x)) 5
- Reduce: (λx.λy.λz.y+3) 4 5 6 =

Simple examples

- Reduce (λx.(2 + x)) 5 → (2 + 5) → 7
- Reduce: (λx.λy.λz.y+3) 4 5 6 =

Simple examples

- Reduce (λx.(2 + x)) 5 → (2 + 5) → 7
- Reduce: (λx.λy.λz.y+3) 4 5 6 =

  (((λx.λy.λz.y+3) 4) 5) 6 →

  ((λy.λz.y+3) 5) 6 →

  (λz.5+3) 6 → (5+3) →8

Simple examples (cont)

- Reduce (λx.(λy.2) 3) 5
- Reduce: ((λx.(λy.2)) 3) 5
- Is (λx.(λy.2) 3) 5 = ((λx.(λy.2)) 3) 5 ?
  - A: yes, B: no

Simple examples (cont)

- Reduce (λx.(λy.2) 3) 5 → (λx. 2) 5 → 2

- Reduce: ((λx.(λy.2)) 3) 5

- Is (λx.(λy.2) 3) 5 = ((λx.(λy.2)) 3) 5 ?

  - A: yes, B: no

Simple examples (cont)

- Reduce (λx.(λy.2) 3) 5 → (λx. 2) 5 → 2

- Reduce: ((λx.(λy.2)) 3) 5 → ((λy.2) 5) → 2

- Is (λx.(λy.2) 3) 5 = ((λx.(λy.2)) 3) 5 ?

  - A: yes, B: no

Simple examples (cont)

- Reduce (λx.(λy.2) 3) 5 → (λx. 2) 5 → 2

- Reduce: ((λx.(λy.2)) 3) 5 → ((λy.2) 5) → 2

- Is (λx.(λy.2) 3) 5 = ((λx.(λy.2)) 3) 5 ?

  - A: yes, B: no

A more complicated example

A more complicated example

- Reduce the following:
  - (λx. (λa. x + a) 7) 4

A more complicated example

- Reduce the following:
  - (λx. (λa. x + a) 7) 4
  - (λa. 4 + a) 7

A more complicated example

- Reduce the following:
  - (λx. (λa. x + a) 7) 4
  - (λa. 4 + a) 7
  - 4 + 7

A more complicated example

- Reduce the following:
  - (λx. (λa. x + a) 7) 4
  - (λa. 4 + a) 7
  - 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)
  - (λx. (λa. x + a) 7) (a + 5)

Let’s make this even more fun!

- Instead of 4, let’s apply function to (a + 5)
  - (λx. (λa. x + a) 7) (a + 5)
  - (λa. (a + 5) + a) 7

Let's make this even more fun!

- Instead of 4, let's apply function to (a + 5)
  - (λx. (λa. x + a) 7) (a + 5)
  - (λa. (a + 5) + a) 7
  - (7 + 5) + 7

Let’s make this even more fun!

- Instead of 4, let’s apply function to (a + 5)
  - (λx. (λa. x + a) 7) (a + 5)
  - (λa. (a + 5) + a) 7
  - (7 + 5) + 7
  - 19

Let’s make this even more fun!

- Instead of 4, let’s apply function to (a + 5)
  - (λx. (λa. x + a) 7) (a + 5)
  - (λa. (a + 5) + a) 7
  - (7 + 5) + 7
  - 19

Is this right?

A: yes, B: no

Let’s make this even more fun!

- Instead of 4, let’s apply function to (a + 5)
  - (λx. (λa. x + a) 7) (a + 5)
  - (λa. (a + 5) + a) 7
  - (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
  - ((λf.(λx. f (f x))) (λy.y+1)) 4

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+1)) 4

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+1)) 4
  -
  -

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+1)) 4
  -
  -
  -

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+1)) 4
  -
  -
  -
  -

Example to do at home

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+x)) 4

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+x)) 4

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+x)) 4
  -
  -

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+x)) 4
  -
  -
  -

Example to do at home

- Reduce the following
  - ((λf.(λx. f (f x))) (λy.y+x)) 4
  -
  -
  -
  -

Another way to see the problem

- Syntactic sugar: let x = e_1 in $e_2 \stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1$

- Let syntax makes this easy to see:

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

  → let x = a + 5 in
  let a = 7 in
  (a + 5) + a

Another way to see the problem

- Syntactic sugar: let x = e_1 in $e_2 \stackrel{\text{def}}{=} (\lambda x.e_2)\ e_1$

- Let syntax makes this easy to see:

  - let x = a + 5 in
    let a = 7 in
    x + a
    → let x = a + 5 in
    let a = 7 in
    (a + 5) + a

  - 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 → 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 a = 7 in
  x + a

→

let x = a+5 in
  let a123 = 7 in
  x + a123

→

let x = a+5 in
  let a123 = 7 in
  (a+5) + a123

2. Do the "dumb" substitution!

Why is this the way to go?

- We can always rename bound variables!
  - Def: variable x is bound in λx.(x+y)
- Bound variables are just "placeholders"
  - Above: x is not special, we could have used z
  - We say they are equivalent: λx.(x+y) =α λz.(z+y)
- Renaming amounts to converting bound variable names to avoid capture: e.g., λx.(x+y) to λz.(z+y)

Can we rename everything?

- Can we rename y in λx.(x+y)? (A: yes, B: no)
  - ➤

- Intuition:
  - ➤
  - ➤

Can we rename everything?

- Can we rename y in λx.(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 λx.(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., ∀x. P(x, y) or $\sum_{i \in \{1,\ldots,10\}} x_i + y$

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 λx.(x+y)
  - is x free? A: yes, B: no

- We can compute the free variables of any term:
  - FV(x) = {x}
  - FV(λx.e) = FV(e) \ {x}
  - FV(e_1 e_2) = FV(e_1) ∪ FV(e_2)

think: build out!

Def: free variables

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in λx.(x+y)
  - is x free? A: yes, B: no

- We can compute the free variables of any term:
  - FV(x) = {x}
  - FV(λx.e) = FV(e) \ {x}
  - FV(e_1 e_2) = FV(e_1) ∪ FV(e_2)

think: build out!

Def: free variables

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in λx.(x+y)
  - is x free? A: yes, B: no

- We can compute the free variables of any term:
  - FV(x) = {x}
  - FV(λx.e) = FV(e) \ {x}
  - FV(e_1 e_2) = FV(e_1) ∪ FV(e_2)

think: build out!

Def: free variables

- If a variable is not bound by a λ, we say that it is **free**
  - e.g., y is free in λx.(x+y)
  - is x free? No! We say x is bound in λx.(x+y)

- We can compute the free variables of any term:
  - $FV(x) = \{x\}$
  - $FV(\lambda x.e) = FV(e) \setminus \{x\}$
  - $FV(e_1\ e_2) = FV(e_1) \cup FV(e_2)$

think: build out!

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:
  - (λx. (λa. x + a) 7) (a + 5)

Back to our example

- What we should have done:
  - (λx. (λa. x + a) 7) (a + 5)

=α (λx. (λb. x + b) 7) (a + 5)

Back to our example

- What we should have done:
  - (λx. (λa. x + a) 7) (a + 5)

  $=_\alpha$ (λx. (λb. x + b) 7) (a + 5)

  $=_\beta$ (λb. (a + 5) + b) 7

Back to our example

- What we should have done:
  - (λx. (λa. x + a) 7) (a + 5)

=α (λx. (λb. x + b) 7) (a + 5)

=β (λb. (a + 5) + b) 7

=β (a + 5) + 7

Back to our example

- What we should have done:
  - (λx. (λa. x + a) 7) (a + 5)

=α (λx. (λb. x + b) 7) (a + 5)

=β (λb. (a + 5) + b) 7

=β (a + 5) + 7

=β a + 12

The to do at home example

The to do at home example

- What you should have done:
  - (λf.(λx. f (f x)) (λy.y+x)

The to do at home example

- What you should have done:
  - (λf.(λx. f (f x)) (λy.y+x)
  - =α (λf.(λz. f (f z)) (λy.y+x)

The to do at home example

- What you should have done:
  - (λf.(λx. f (f x)) (λy.y+x)

=α (λf.(λz. f (f z)) (λy.y+x)

=β λz. (λy.y+x) ((λy.y+x) z)

The to do at home example

- What you should have done:
  - (λf.(λx. f (f x)) (λy.y+x)

=α (λf.(λz. f (f z)) (λy.y+x)

=β λz. (λy.y+x) ((λy.y+x) z)

=β λz. (λy.y+x) (z+x)

The to do at home example

- What you should have done:
  - (λf.(λx. f (f x)) (λy.y+x)

=α (λf.(λz. f (f z)) (λy.y+x)

=β λz. (λy.y+x) ((λy.y+x) z)

=β λz. (λy.y+x) (z+x)

=β λz. z+x+x

Week 2

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