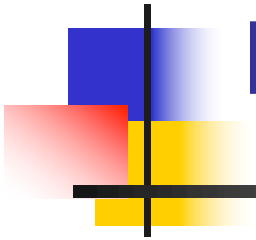


Lambda Calculus

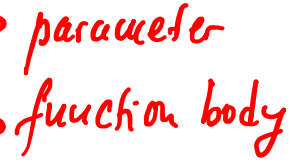
Reading: Scott, Ch. 11.7 (on Companion Site)

Lecture Outline

- *Lambda calculus*
  - *Introduction*
  - *Syntax and semantics*
  - *Free and bound variables*
  - *Substitution, formally*

Lambda Calculus

- A theory of functions
  - Theory behind functional programming
  - Turing complete: any computable function can be expressed and evaluated using the calculus
  - "Lingua franca" of PL research
- Lambda (λ) calculus expresses function definition and function application
  - **f(x)=x*x** becomes **λx. x*x**
  - **g(x)=x+1** becomes **λx. x+1**
  - **f(5)** becomes **(λx. x*x) 5 → 5*5 → 25**



Syntax of Pure Lambda Calculus

Convention:
notation f, x, y, z for variables;
E, M, N, P, Q for expressions

- **E ::= x | (λx. E_1) | (E_1 E_2)**
  - A λ-expression is one of
    - Variable: **x**
    - Abstraction (i.e., function definition): **λx. E_1**
    - Application: **E_1 E_2**

- λ-calculus formulae (e.g., **(λx. (x y))**) are called expressions or terms
- **(λx. (x y))** corresponds to (lambda (x) (x y)) in Scheme!

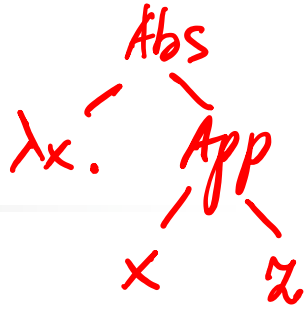
Syntactic Conventions

- Parentheses may be dropped from **(E_1 E_2)** or **(λx.E)**
  - E.g., **(f x)** may be written as **f x**

- Function application groups from left-to-right (i.e., it is left-associative)
  - E.g., **x y z** abbreviates **((x y) z)**
  - E.g., **E_1 E_2 E_3 E_4** abbreviates **(((E_1 E_2) E_3) E_4)**
  - Parentheses in **x (y z)** are necessary! Why?

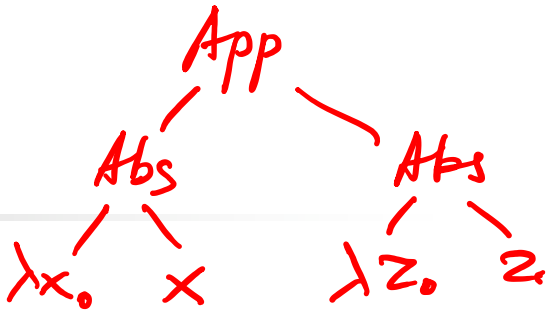
x y z ≡ (x y) z ≠ x (y z)

Syntactic Conventions



- Application has higher precedence than abstraction
  - Another way to say this is that the scope of the dot extends as far to the right as possible
  - E.g., **λx. x z ≅ λx. (x z) ≅ (λx. (x z))**
    ≇ ((λx. x) z)
- WARNING: This is the most common syntactic convention (e.g., Pierce 2002). Some books give abstraction higher precedence

Terminology



- Parameter (also, formal parameter)
  - E.g., **x** is the parameter in **λx. x z**

- Argument (also, actual argument)
  - E.g., expression **λz. z** is the argument in

(λx. x) (λz. z) → λz.z

Can you guess what this evaluates to?

E_1 E_2

Currying

- In the lambda calculus, all functions have one parameter
  - How do we express n-ary functions?
  - Currying expresses an n-ary function in terms of n unary functions

Takes x and returns the plus-x function.

f(x,y) = x+y, becomes **(λx.λy. x + y)**

(λx.λy. x + y) 2 3 → (λy. 2 + y) 3 → 2 + 3 → 5

plus-2

Currying in Scheme

```
(define curried-plus
    (lambda (a) (lambda (b) (+ a b))))
```

- `(curried-plus 3)` returns what?
  - Returns the plus-three function (or more precisely, it returns a closure)

- `((curried-plus 3) 2)` returns what?

Currying

$f(x_1, x_2, \ldots, x_n) = g\ x_1\ x_2 \ldots x_n$

$g_1\ x_2$

$g_2\ x_3$

...

Function **g** is said to be the curried form of **f**

Semantics of Pure Lambda Calculus

- An expression has as its meaning the value that results after evaluation is carried out
  - Somewhat informally, evaluation is the process of reducing expressions

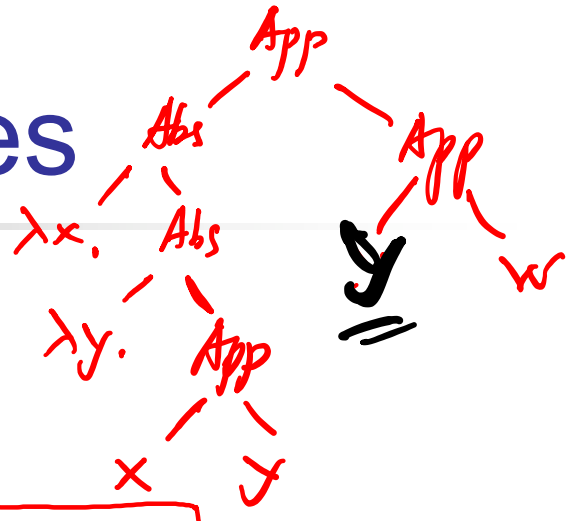
E.g., **(λx.λy. x + y) 3 2 → (λy. 3 + y) 2 → 3 + 2 → 5**

(Note: this example is just an informal illustration. There is no **+** in the pure lambda calculus!)

Lecture Outline

- *Lambda calculus*
  - *Introduction*
  - *Syntax and semantics*
  - *Free and bound variables*
  - *Substitution, formally*

Free and Bound Variables



- Reducing expressions

- Consider expression **(λx.λy. x y) (y w)**
- Try 1: Reducing results in the following

  (λy. x y) [(y w)/x] = (λy. (y w) y)

  Above notation means: we substitute argument **(y w)** for every occurrence of parameter **x** in body **(λy. x y)**.

  But what is wrong here?

- **(λx.λy. x y) (y w)**: different y's! If we substitute (y w) for x, the "free" y will become "bound"!

Free and Bound Variables

- Try 2: Rename "bound" **y** in **λy. x y** to **z: λz. x z**

  (λx.λy. x y) (y w) => (λx.λz. x z) (y w)

  - E.g., in C, **int id(int p) { return p; }** is exactly the same as **int id(int q) { return q; }**

  - Applying the reduction rule results in

(λz. x z) [(y w)/x] => (λz. (y w) z)

Free and Bound Variables

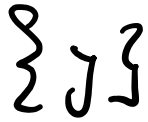
- Abstraction **(λx. E)** is also referred as binding

- Variable **x** is said to be bound in **λx. E**
- The set of free variables of **E** is the set of variables that are unbound in **E**
- Defined by cases on **E**
  - Var **x**: free(**x**) = {**x**}
  - App $\mathbf{E_1}\ \mathbf{E_2}$: $free(\mathbf{E_1}\ \mathbf{E_2}) = free(\mathbf{E_1}) \cup free(\mathbf{E_2})$
  - Abs **λx.E**: free(**λx.E**) = free(**E**) - {**x**}

Free and Bound Variables

- A variable **x** is bound if it is in the scope of a lambda abstraction: as in **λx. E**
- Variable is free otherwise

1. (λx. x) y



2. (λz. z z) (λx. x)

3. λx.λy.λz. x z (y (λu. u))

Free and Bound Variables

- **λx.λy.λz. x z (y (λu. u))**

Free and Bound Variables

- We must take free and bound variables into account when reducing expressions

  E.g., **(λx.λy. x y) (y w)**

  - Reduction rule defined in terms of substitution:

  (λy. x y) [(y w)/x]

  - First, rename bound **y** in **λy. x y** to **z**: **λz. x z**
    (more precisely, we have to rename to a variable that is NOT free in either **(y w)** or **(x y)**)
  - Second, replace **x** with argument **(y w)** safely:
    (λz. (y w) z) = λz. y w z

Lecture Outline

- *Lambda calculus*
  - *Introduction*
  - *Syntax and semantics*
  - *Free and bound variables*
  - *Substitution, formally*

Substitution, formally

- **(λx.E) M → E[M/x]** replaces all free occurrences of **x** in **E** by **M**
- **E[M/x]** is defined by cases on **E**:
  - Var: **y[M/x] = M** if **x = y**

    y[M/x] = y otherwise
  - App: **$(E_1\ E_2)[M/x] = (E_1[M/x]\ E_2[M/x])$**
  - Abs: **$(\lambda y.E_1)[M/x] = \lambda y.E_1$** if **x = y**

    $(\lambda y.E_1)[M/x] = \lambda z.((E_1[z/y])[M/x])$ otherwise,

    where **z** NOT in free(E_1) U free(**M**) U {**x**}

Substitution, formally

(λx.λy. x y) (y w)

- → **(λy. x y)[(y w)/x]**
- → **λ1_. (((x y)[1_/y])[(y w)/x])**
- → **λ1_. ((x 1_)[(y w)/x])**
- → **λ1_. ((y w) 1_)**
- → **λ1_. y w 1_**

Substitution, formally

(λx.λy. λz. x z (y z)) (λx.x)

The End