

Lambda Calculus

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University

Semester 2, 2009

Specifying Functions

In C/C++

```
int max (int n, int m)
{
    if (m > n) return m;
    else      return n;
}
```

```
void main (void) {
    cout << max (2, 3);
}
```

In Haskell

```
max :: Int -> Int -> Int
max m n | m > n      = m
        | otherwise = n
```

```
main :: IO ()
main = print (max 2 3)
```

What are programs made from?

- The previous two definitions are very similar.
- What features are common to most languages?

Variable names	Function parameters	Pattern matching
Function bodies	Function application	Sequencing
Substitution	Choice	Exceptions
Data structures	State	Parallelism
Data types	Primitive operations	Templates
Objects	Polymorphism	Reflection
Input / Output	Operator Overloading	Multiple Inheritance

Simplify!

Try and imagine the smallest possible programming language.

What features would it absolutely*, positively*, have to have?

- Variable names
- Functions
- Function application

* lies

Lambda Notation

Shift function parameters to the right of the =.

$$f\ x = x^2 + x + 1$$

$$f = \lambda x. x^2 + x + 1$$

$$g\ x\ y = x + y$$

$$g = \lambda x. \lambda y. x + y$$

f and g are the same functions, only the notation to define them has changed.

We can also define *anonymous functions*, that don't have names:

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

The (pure) Lambda Calculus

Variable names

x, y, z, \dots

Expressions

e	$=$	x	(variable)
		$\lambda x. e$	(function abstraction)
		$e e$	(function application)

Evaluation

$$(\lambda x. e_1) e_2 \longrightarrow e_1[x := e_2]$$

Evaluation (also called β -reduction or β -conversion)

$$(\lambda x. e_1) e_2 \longrightarrow e_1[x := e_2]$$

x is the *formal parameter*

e_1 is the *function body*

e_2 is the *function argument*

$e_1[x := e_2]$ means: “ e_1 , with e_2 substituted for x ”

The substitution can also be written $e_1[e_2/x]$

λ

All computable functions can be expressed
in the lambda calculus.

λ Substitution is all you need! λ

Example Evaluations

Choose a reducible expression and substitute the argument into the function body. A reducible expression is also called a *redex*.

$$\begin{aligned} & (\lambda x. x y) \left(\underline{(\lambda z. z) u} \right) \\ & \longrightarrow \underline{(\lambda x. x y) u} \\ & \longrightarrow u y \end{aligned}$$

$$\begin{aligned} & \underline{(\lambda x. \lambda y. y x x) a b c} \\ & \longrightarrow \underline{(\lambda y. y a a) b c} \\ & \longrightarrow b a a c \end{aligned}$$

When the expression can be reduced no further it is in *normal form*.

Syntactic Conventions

Function abstraction associates to the right:

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

Function application associates to the left:

$$f g h x \equiv ((f g) h) x$$

Sequences of λ s may be collapsed:

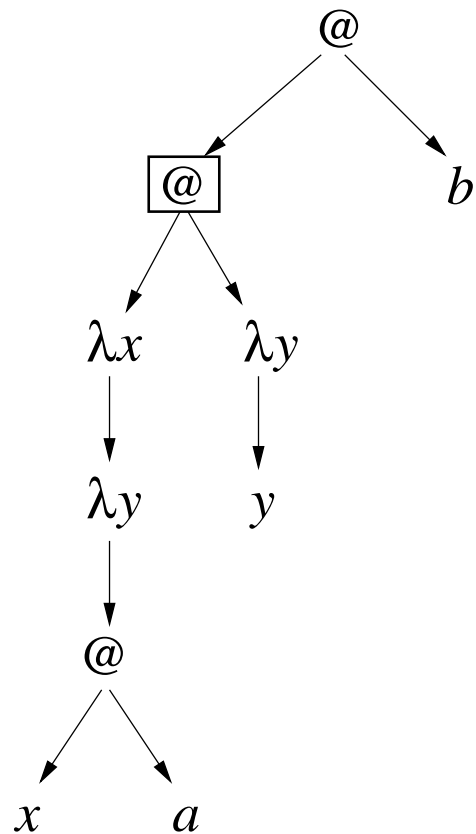
$$\lambda x. \lambda y. \lambda z. e \equiv \lambda x y z. e$$

Application has precedence over abstraction:

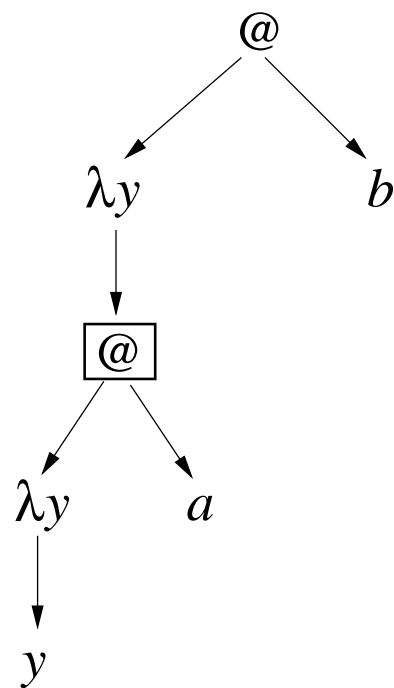
$$\lambda x. e_1 e_2 \equiv \lambda x. (e_1 e_2)$$

Abstract Syntax Trees

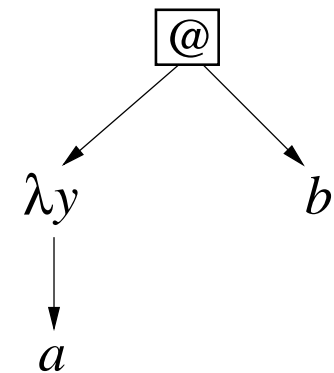
$(\lambda x. \lambda y. x a) (\lambda y. y) b$



$(\lambda y. (\lambda y. y) a) b$

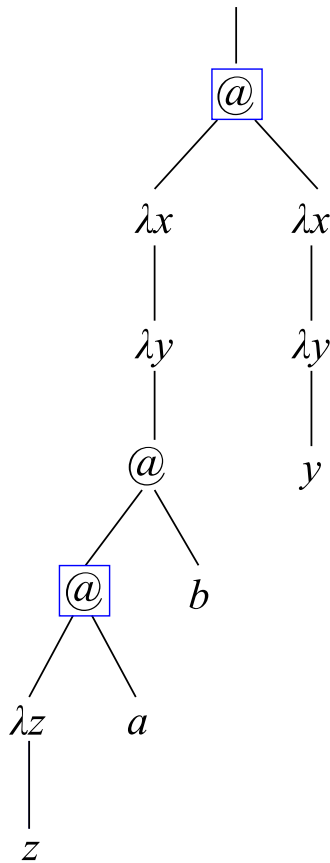


$(\lambda y. a) b$



There can be multiple redexes in an expression

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



- Note that a redex is an $@$ (application) node with a λ on the left.
- We could choose either redex as the next one to reduce.

Confluence

$$\begin{aligned} & (\lambda x. x y) \left(\underline{((\lambda z. z) u)} \right) \\ & \longrightarrow \underline{(\lambda x. x y) u} \\ & \longrightarrow u y \end{aligned}$$

$$\begin{aligned} & \underline{(\lambda x. x y) \left(\underline{((\lambda z. z) u)} \right)} \\ & \longrightarrow \underline{((\lambda z. z) u) y} \\ & \longrightarrow u y \end{aligned}$$

Each expression has at most one normal form.

Reduction order does not matter*.

The lambda calculus is *confluent*

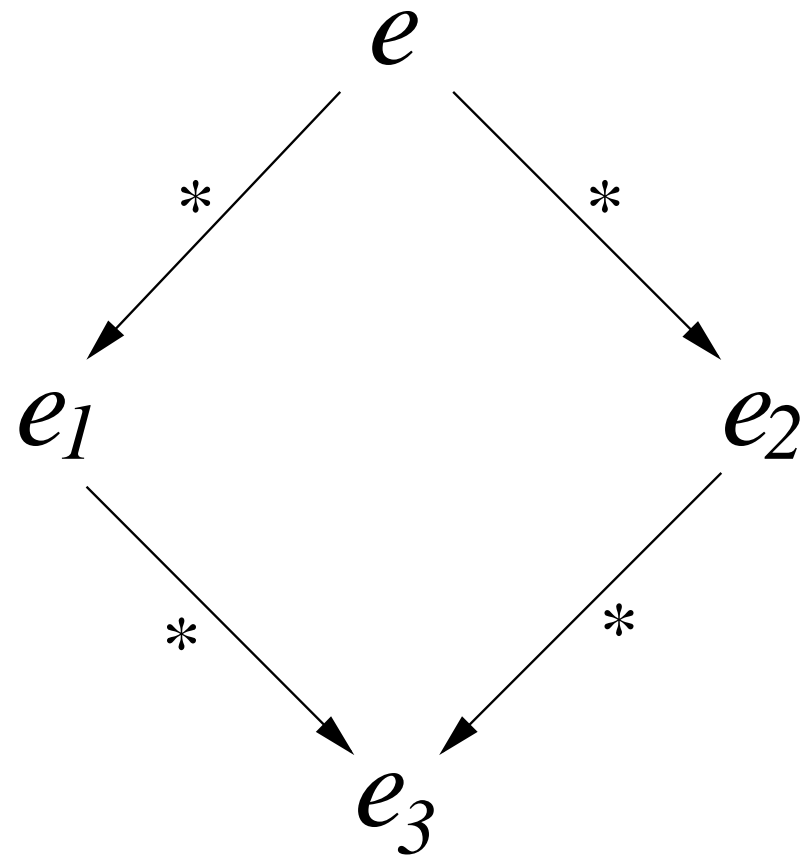
* modulo divergence and implementation efficiency



Photo: Morten Oddvik (CCA2.0)

The Church-Rosser Theorem expresses Confluence

If $e \xrightarrow{*} e_1$ and $e \xrightarrow{*} e_2$
then there is some e_3 such that
 $e_1 \xrightarrow{*} e_3$ and $e_2 \xrightarrow{*} e_3$



$a \xrightarrow{*} b$ means b can be reduced
from a , in some (possibly null) succession of steps.

Variable Capture

Consider this expression:

$$\begin{aligned} & \underline{(\lambda x. \lambda y. x a) (\lambda y. y) b} \\ & \longrightarrow (\lambda y. \underline{(\lambda y. y) a}) b \\ & \longrightarrow \underline{(\lambda y. a) b} \\ & \longrightarrow a \end{aligned}$$

Using a different evaluation order breaks confluence... ??

$$\begin{aligned} & \underline{(\lambda x. \lambda y. x a) (\lambda y. y) b} \\ & \longrightarrow \underline{(\lambda y. (\lambda y. y) a) b} \\ & \longrightarrow (\lambda y. b) a \\ & \longrightarrow b \end{aligned}$$

Free, Bound and Binding Variables

In the expression $(\lambda x. x y)$

- the variable y is *free* because there is no enclosing λy
- the first x is the *binding* occurrence.
- the second x is a *bound* occurrence.
- an expression with no free variables is *closed*.

A function to compute the free variables in an expression:

$$\text{fv}(x) = \{ x \}$$

$$\text{fv}(\lambda x. e) = \text{fv}(e) - \{x\}$$

$$\text{fv}(e_1 e_2) = \text{fv}(e_1) \cup \text{fv}(e_2)$$

In examples I usually use x, y, z for bound variables and a, b, c for free variables, but that's just my own convention.

Expression	Bound Vars	Free Vars
$\lambda x. x$	$\{x\}$	\emptyset
$\lambda x. \lambda y. x$	$\{x, y\}$	\emptyset
a	\emptyset	$\{a\}$
$a (b c)$	\emptyset	$\{a, b, c\}$
$\lambda x. a$	$\{x\}$	$\{a\}$
$(\lambda x. \lambda y. x y) (\lambda z. a)$	$\{x, y, z\}$	$\{a\}$

α -conversion

A function's behavior should be independent of the name of its bound variable. Renaming bound variables is called *α -conversion*.
(read: α -conversion == name-conversion)

The identity function always returns its argument:

$$(\lambda x. x) u \longrightarrow u$$

Renaming the bound variable yields the same result:

$$(\lambda y. y) u \longrightarrow u$$

Inside out

A bound variable belongs to the inner-most binding occurrence:

We can safely *α -convert* the expression:

$$\lambda x. \lambda x. x$$

to

$$\lambda x. \lambda y. y$$

eg: $(\lambda x. \lambda y. y) u \longrightarrow (\lambda y. y)$

but not

$$\lambda y. \lambda x. y$$

eg: $(\lambda y. \lambda x. y) u \longrightarrow (\lambda x. u)$

Two expressions which differ only in the names of the bound variables are said to be *α -equivalent*.

To avoid variable capture

α -convert the initial expression so that binding occurrences of variables have unique names.

$$\begin{aligned} & \underline{(\lambda x. \lambda y. x a) (\lambda y. y) b} \\ & \longrightarrow \underline{(\lambda y. (\lambda y. y) a) b} \\ & \longrightarrow \underline{(\lambda y. y) a} \\ & \longrightarrow a \end{aligned}$$

$$\begin{aligned} & \underline{(\lambda x. \lambda y. x a) (\lambda y. y) b} \\ & \xrightarrow{\alpha} \underline{(\lambda x. \lambda y. x a) (\lambda z. z) b} \\ & \longrightarrow \underline{(\lambda y. (\lambda z. z) a) b} \\ & \longrightarrow (\lambda z. z) a \\ & \longrightarrow a \end{aligned}$$

Renaming variables to have unique names is often one of the first stages in the compiler for a functional language.

Once renamed, we can reduce and transform an arbitrary expression without worrying about variable capture.

Alternatively: Use capture avoiding substitution

$$x [x := e_1] = e_1$$

$$y [x := e_1] = y$$

$$(e_2 e_3)[x := e_1] = (e_2[x := e_1]) (e_3[x := e_1])$$

Stop when the bound variable matches the one being substituted for.

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

If the bound var isn't free in the new expr then we're ok.

$$(\lambda y. e_2)[x := e_1] = \lambda y. e_2[x := e_1] \quad \text{when } y \notin \text{fv}(e_1)$$

If the bound var is free in the new expr, then α -convert the abstraction.

$$(\lambda y. e_2)[x := e_1] = \lambda z. e_2[y := z][x := e_1] \quad \text{when } y \in \text{fv}(e_1)$$

z is a fresh variable.

Worrying about variable capture is tedious and annoying.

- We could argue that it's just an artefact of our textual representation of the lambda calculus, and not a property of the deeper system.
- For concrete implementations, it's easier to rename all bound variables, or give them unique identifiers.
- Literature usually states that the substitution operator is capture avoiding — or states that equivalence of λ -expressions is “up to α -conversion”.

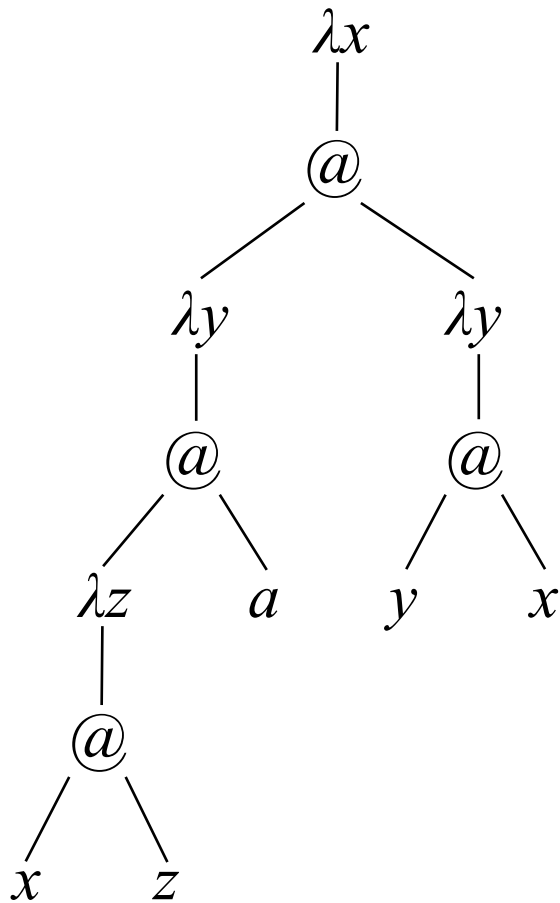
De Bruijn indices

- Alternatively, to avoid problems with variable capture we can replace variables by de Bruijn indices.
- The value of the index is the number of λ s up the tree the binding λ is.
- This eliminates problems with the substitution, but during β -reduction we must adjust indices in the function argument
- It's an alternate scheme, but not a clear win for all applications.

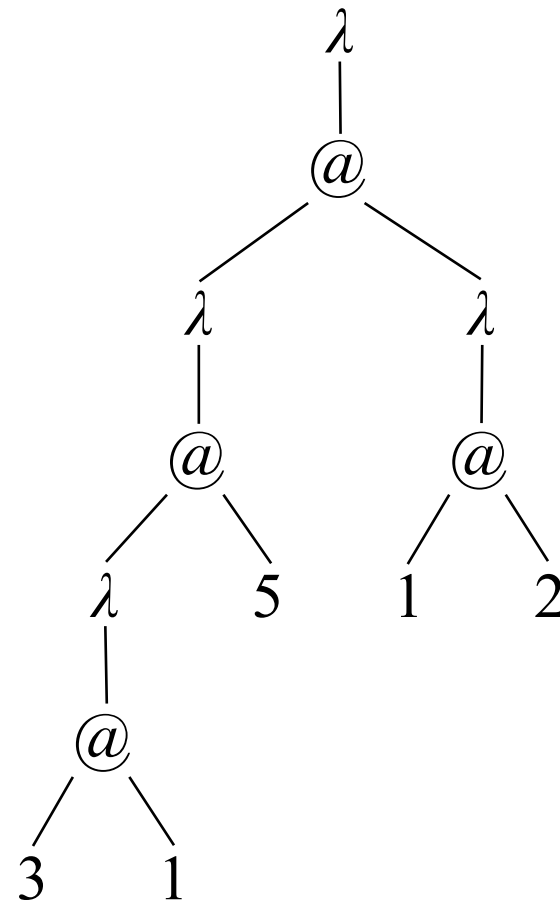
De Bruijn indices example

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

$\lambda (\lambda (\lambda 3 1) 5) (\lambda 1 2)$



\equiv



η -conversion (η = “eta”)

The function that takes any a to $f a$ is just the function f .

η -conversion is the process of adding (or removing) new λ -abstractions around functions, or function variables.

Consider:

$$f a \xrightarrow{\eta} (\lambda x. f x) a \xrightarrow{\eta} (\lambda y. (\lambda x. f x) y) a$$

We can always β -reduce to get the original expression.

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

Divergence

$$\underline{(\lambda x. x x) (\lambda x. x x)}$$
$$\longrightarrow \underline{(\lambda x. x x) (\lambda x. x x)}$$
$$\longrightarrow \underline{(\lambda x. x x) (\lambda x. x x)}$$
$$\longrightarrow \dots$$
$$\underline{(\lambda x. x x x) (\lambda x. x x x)}$$
$$\longrightarrow \underline{(\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)}$$
$$\longrightarrow \underline{(\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x) (\lambda x. x x x)}$$
$$\longrightarrow \dots$$

Both of these expressions *diverge*.

Neither can be reduced to *normal form*.

The (untyped) lambda calculus is not *normalising*.

Evaluation order

The choice of evaluation order affects which expressions diverge.

Call-by-name: Substitute arguments directly into function bodies. If an argument is not used in the function body it is never evaluated.

$$\begin{aligned} & \underline{(\lambda y. a) ((\lambda x. x x) (\lambda x. x x))} \\ & \longrightarrow a \end{aligned}$$

Call-by-value: Evaluate arguments to normal form before substituting into function bodies. If an argument diverges, so will the whole expression.

$$\begin{aligned} & (\lambda y. a) \underline{((\lambda x. x x) (\lambda x. x x))} \\ & \longrightarrow (\lambda y. a) \underline{((\lambda x. x x) (\lambda x. x x))} \\ & \longrightarrow (\lambda y. a) \underline{((\lambda x. x x) (\lambda x. x x))} \\ & \longrightarrow \dots \end{aligned}$$

Derived Forms

It's sometimes nice to give useful expressions names, eg:

let *id* = $\lambda x. x$ **in**

let *fst* = $\lambda x. \lambda y. x$ **in**

let *snd* = $\lambda x. \lambda y. y$ **in**

in *id* (*snd* *a* *b*)

This is nothing more than substitution, which we already have.

$$\mathbf{let } x = e_1 \mathbf{ in } e_2 \stackrel{\text{def}}{=} (\lambda x. e_2) e_1$$