

# Lambda Calculus

## COMP2600 — Formal Methods for Software Engineering

Jeremy Dawson

Australian National University  
Semester 2, 2011

## 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

## 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)
```

## Simplify!

Try and imagine the smallest possible programming language.

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

- Variable names
- Functions
- Function application

\*lies, actually

## The Lambda Calculus

### Variable names

$x, y, z, \dots$

### Expressions

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

### Evaluation

$(\lambda x. e_1) e_2 \longrightarrow e_1[e_2/x]$   $e_1$ , with occurrences of  $x$  replaced by  $e_2$

$(\lambda x. e_1)$  means the function that takes an argument, calls it  $x$ , and returns  $e_1$

## Example Evaluations

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

Make sure to pair the correct lambda with each argument. Add the parenthesis back in if you're not sure.

$$\begin{array}{ll}
 (\lambda x. x y) ((\lambda z. z) u) & (\lambda x. \lambda y. y x x) a b c \\
 \longrightarrow (\lambda x. x y) u & \longrightarrow (\lambda y. y a a) b c \\
 \longrightarrow u y & \longrightarrow b a a c
 \end{array}$$

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

$\lambda$

All computable functions can be expressed  
in the lambda calculus.

$\lambda$  Substitution is all you need!  $\lambda$

## 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  $\lambda$ s may be collapsed:

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

Application has precedence over abstraction:

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

## Evaluation ( $\beta$ -reduction)

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

$x$  is the *formal parameter* or *bound variable*

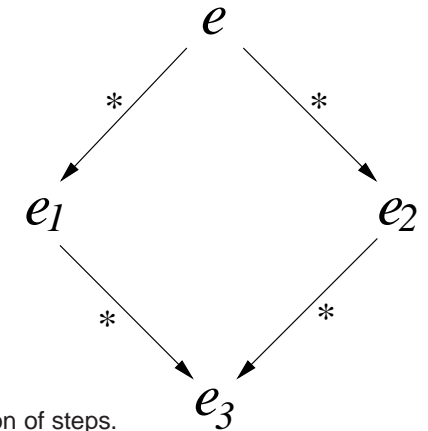
$e_1$  is the *function body*

$e_2$  is the *function argument*

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

## The Church-Rosser Theorem

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.

## Confluence

$$\begin{array}{l} (\lambda x. x y) ((\lambda z. z) u) \\ \longrightarrow (\lambda x. x y) u \\ \longrightarrow u y \end{array} \qquad \begin{array}{l} (\lambda x. x y) ((\lambda z. z) u) \\ \longrightarrow ((\lambda z. z) u) y \\ \longrightarrow u y \end{array}$$

Each expression has at most one normal form.  
 Reduction order does not matter.  
 The lambda calculus is *confluent*



Photo: Morten Oddvik (CCA2.0)

## Variable Capture

Consider this expression:

$$\begin{array}{l} (\lambda x. \lambda y. x a) (\lambda y. y) b \longrightarrow (\lambda y. (\lambda y. y) a) b \\ \longrightarrow (\lambda y. a) b \\ \longrightarrow a \end{array}$$

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

$$\begin{array}{l} (\lambda x. \lambda y. x a) (\lambda y. y) b \longrightarrow (\lambda y. (\lambda y. y) a) b \\ \longrightarrow (\lambda y. b) a \quad \text{WRONG!} \\ \longrightarrow b \end{array}$$

WRONG as  $\lambda y. (\lambda y. y) a$  is  $\lambda y_1. (\lambda y_2. y_2) a$

## Free, Bound and Binding Variables

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

- the variable  $y$  is *free* because there is no enclosing  $\lambda y$ 
  - the expression is undefined until  $y$  is given a definition
  - but  $x$  is “defined” by its occurrence in  $\lambda x$ ; any meaning it had outside the expression  $(\lambda x. x y)$  is hidden (“shadowed”)
- the first  $x$  is the *binding* occurrence.
- the second  $x$  is a *bound* occurrence.
- an expression with no free variables is *closed*.

## $\alpha$ -conversion

A function’s behavior should be independent of the name of its bound variable. Renaming the bound variable is called  *$\alpha$ -conversion*.

The identity function always returns its argument:

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

Renaming the bound variable yields the same result:

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

## A function to compute the free variables in an expression

$$\begin{aligned} \text{fv}(x) &= \{ x \} \\ \text{fv}(\lambda x. e) &= \text{fv}(e) - \{ x \} \\ \text{fv}(e_1 e_2) &= \text{fv}(e_1) \cup \text{fv}(e_2) \end{aligned}$$

## Inside out

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

Thus  $\lambda x. (\lambda x. x)$  is  $\lambda x_1. (\lambda x_2. x_2)$

We can safely  $\alpha$ -convert:

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

to

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

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

but not

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

$$\text{eg: } (\lambda y. \lambda x. y) u \longrightarrow (\lambda x. u) \quad \text{WRONG!!}$$

## Capture avoiding substitution

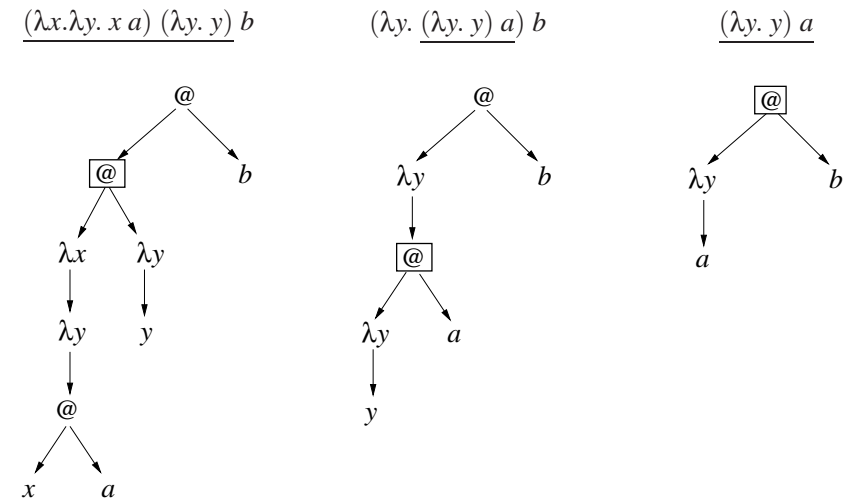
When substituting for  $y$ , stop when we reach another binding occurrence of  $y$ .

**Equivalently:**  $\alpha$ -convert the initial expression so that binding occurrences of variables have unique names.

$$\begin{array}{l} \underline{(\lambda x. \lambda y. x a) (\lambda y. y) b} \\ \longrightarrow \underline{(\lambda y. (\lambda y. y) a) b} \\ \longrightarrow \underline{(\lambda y. y) a} \\ \longrightarrow a \end{array} \qquad \begin{array}{l} \underline{(\lambda x. \lambda y. x a) (\lambda y. y) b} \\ \xrightarrow{\alpha} \underline{(\lambda x. \lambda y. x a) (\lambda z. z) b} \\ \longrightarrow \underline{(\lambda y. (\lambda z. z) a) b} \\ \longrightarrow (\lambda z. z) a \\ \longrightarrow a \end{array}$$

Two expressions which differ only in the names of the bound variables are said to be  **$\alpha$ -equivalent**. eg:  $(\lambda y. y)$  and  $(\lambda z. z)$ .

## Expression Trees



## Variable Capture — another situation

To  $\beta$ -reduce the following expression:

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

we *must* first  $\alpha$ -convert the  $\lambda y. x$

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

Without the  $\alpha$ -conversion we would get

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

which is *different*. Here the two 'y's of the original, *which are distinct*, have accidentally been identified (the  $y$  substituted has been "captured")

In  $(\lambda x. \lambda y. x) y$  the two 'y's are unrelated, because inside the  $(\lambda y. x)$  the outer 'y' is hidden, as  $y$  is redefined by the binding  $\lambda y$ .

## $\eta$ -conversion ( $\eta$ = "eta")

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

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

But  $x$  **must NOT** appear free in  $f$  (otherwise we have  $x$  free on the left, but not on the right — which **CAN'T** be right).

**$\eta$ -conversion** is the process of adding (or removing) new  $\lambda$ -abstractions around functions, or function variables, changing  $f$  to  $\lambda x. f x$  or vice-versa.

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

With an argument  $a$  we can *either*  $\beta$ -reduce *or*  $\eta$ -reduce to get back to  $f a$ .

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

## Divergence

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

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

Both of these expressions *diverge*.

Neither can be reduced to *normal form*.

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

## Derived Forms

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

$$\begin{aligned} \mathbf{let} \quad id &= \lambda x. x \quad \mathbf{in} \\ \mathbf{let} \quad fst &= \lambda x. \lambda y. x \quad \mathbf{in} \\ \mathbf{let} \quad snd &= \lambda x. \lambda y. y \quad \mathbf{in} \\ &id (snd a b) \end{aligned}$$

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

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

## 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.

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

## All computable functions?

- We have seen variables, functions, and substitution.
- What about numbers, tuples and lists?
- These ideas can also be encoded as derived forms.
- Our *fst* and *snd* functions are already half way there.

## Selecting from a pair

We saw these functions before

$$\text{fst} \stackrel{\text{def}}{=} \lambda x y. x$$

$$\text{snd} \stackrel{\text{def}}{=} \lambda x y. y$$

$\text{fst } \text{cat } \text{dog}$

$$\begin{aligned} &\xrightarrow{\text{def}} \underline{(\lambda x y. x) \text{ cat } \text{dog}} \\ &\longrightarrow \underline{(\lambda y. \text{cat}) \text{ dog}} \\ &\longrightarrow \text{cat} \end{aligned}$$

$\text{snd } \text{cat } \text{dog}$

$$\begin{aligned} &\xrightarrow{\text{def}} \underline{(\lambda x y. y) \text{ cat } \text{dog}} \\ &\longrightarrow \underline{(\lambda y. y) \text{ dog}} \\ &\longrightarrow \text{dog} \end{aligned}$$

## Church Numerals

Logically, we define natural numbers inductively:

- **zero** is a natural number.
- if  $n$  is a natural number then (**succ**  $n$ ) is also a natural number.

$$0 = \text{zero}$$

$$1 = \text{succ zero}$$

$$2 = \text{succ (succ zero)}$$

$$3 = \text{succ (succ (succ zero))}$$

$$4 = \dots$$

But how do we define **succ** and **zero**?

## Booleans and choice

A Boolean value expresses a choice between two options:

$$\text{true} \stackrel{\text{def}}{=} \lambda x y. x$$

$$\text{false} \stackrel{\text{def}}{=} \lambda x y. y$$

$\text{true } \text{cat } \text{dog}$

$$\begin{aligned} &\xrightarrow{\text{def}} \underline{(\lambda x y. x) \text{ cat } \text{dog}} \\ &\longrightarrow \underline{(\lambda y. \text{cat}) \text{ dog}} \\ &\longrightarrow \text{cat} \end{aligned}$$

$\text{false } \text{cat } \text{dog}$

$$\begin{aligned} &\xrightarrow{\text{def}} \underline{(\lambda x y. y) \text{ cat } \text{dog}} \\ &\longrightarrow \underline{(\lambda y. y) \text{ dog}} \\ &\longrightarrow \text{dog} \end{aligned}$$

So “ $\text{cond } \text{cat } \text{dog}$ ” (when  $\text{cond}$  is true or false) is like “if  $\text{cond}$  then **cat** else **dog**”

We are more interested in *interaction* than *representation*.

Booleans embody the idea of *choosing between two things*.

Numbers embody the idea of *doing something multiple times*.

We can make the *something* abstract:

$$c_0 \stackrel{\text{def}}{=} \lambda s z. z$$

$$c_1 \stackrel{\text{def}}{=} \lambda s z. s z$$

$$c_2 \stackrel{\text{def}}{=} \lambda s z. s (s z)$$

$$c_3 \stackrel{\text{def}}{=} \dots$$

So we do ‘something’,  $s$ , multiple times, to an ‘initial value’,  $z$ .

$s$  and  $z$  have the role of **succ** and **zero**, but we leave it up to the caller to decide what to use for them

## Addition

$$\mathbf{plus} \equiv \lambda m. \lambda n. \lambda s z. m s (n s z)$$

**plus**  $c_2 c_3$

$$\xrightarrow{\text{def}} (\lambda m. \lambda n. \lambda s z. m s (n s z)) c_2 c_3$$

$$\xrightarrow{*} (\lambda s z. c_2 s (c_3 s z))$$

$$\xrightarrow{\text{def}} (\lambda s z. c_2 s ((\lambda s z. s (s (s z))) s z))$$

$$\xrightarrow{*} (\lambda s z. c_2 s (s (s (s z))))$$

$$\xrightarrow{\text{def}} (\lambda s z. (\lambda s z. s (s z)) s (s (s (s z))))$$

$$\xrightarrow{*} (\lambda s z. (s (s (s (s (s z))))))$$

$$\xrightarrow{\text{def}} c_5$$

## Multiplication

$$\mathbf{times} \equiv \lambda m. \lambda n. \lambda s z. m (n s) z$$

that is,  $\mathbf{times} c_m c_n s = c_m (c_n s) = (s^n)^m$  which of course is  $s^{nm}$

Following slide shows a detailed calculation for **times**  $c_2 c_3$

## Power

$$\mathbf{pow} \equiv \lambda m. \lambda n. n m$$

that is,  $\mathbf{pow} c_m c_n s = (c_m)^n s = \overbrace{c_m (c_m \dots (c_m s) \dots)}^n = ((\dots (s^m)^m \dots)^m)$   
 which of course is  $s^{m^n}$

## A more intuitive notation

$$c_n s = s^n$$

where  $s^n$  is function  $s$  to the  $n$ th power, ie, composed  $n$  times

$$s^n x = \overbrace{s(s \dots (s x) \dots)}^n$$

So what we have just seen is

$$\mathbf{plus} c_m c_n s x = c_m s (c_n s x) = s^m (s^n x)$$

which of course is  $s^{m+n} x$

**times**  $c_2 c_3$

$$\xrightarrow{\text{def}} (\lambda m. \lambda n. \lambda s z. m (n s) z) c_2 c_3$$

$$\xrightarrow{*} (\lambda s z. c_2 (c_3 s) z)$$

$$\xrightarrow{\text{def}} (\lambda s z. c_2 ((\lambda s z. s (s (s z))) s) z)$$

$$\longrightarrow (\lambda s z. c_2 (\lambda z. s (s (s z))) z)$$

$$\xrightarrow{\text{def}} (\lambda s z. (\lambda s z. s (s z)) (\lambda z. s (s (s z))) z)$$

$$\longrightarrow (\lambda s z. (\lambda z. (\lambda z. s (s (s z))) ((\lambda z. s (s (s z))) z)) z)$$

$$\longrightarrow (\lambda s z. (\lambda z. (\lambda z. s (s (s z))) (s (s (s z)))) z)$$

$$\longrightarrow (\lambda s z. (\lambda z. s (s (s (s (s z)))))) z)$$

$$\longrightarrow (\lambda s z. s (s (s (s (s z)))))$$

$$\xrightarrow{\text{def}} c_6$$