

Lambda Calculus for Advanced Programming

Based on Dr MC du Plessis' Lectures

June 2016

Introduction

In Lambda Calculus

- ▶ $\langle \textit{expression} \rangle ::= \langle \textit{name} \rangle \mid \langle \textit{function} \rangle \mid \langle \textit{application} \rangle$
- ▶ $\langle \textit{function} \rangle ::= \lambda \langle \textit{name} \rangle . \langle \textit{expression} \rangle$
- ▶ $\langle \textit{application} \rangle ::= (\langle \textit{function} \rangle \ \langle \textit{expression} \rangle)$

Examples of valid Lambda expressions

- ▶ $\lambda x.x$
- ▶ $\lambda \textit{first}.\lambda \textit{second}.\textit{first}$
- ▶ $\lambda f.\lambda a.(f\ a)$

β -reductions

Everything done when evaluating Lambda expressions are called β -reductions.

Steps: Remove brackets, take away first name and replace with second part of application.

Examples

- ▶ $(\lambda a. \lambda b. (a \ b) \ z) = \lambda b. (z \ b)$
- ▶ $(\lambda x. x \ \lambda y. y) = \lambda y. y$

Identity function example

- ▶ $(\lambda x. x \ \lambda x. x) = \lambda x. x$

More functions

Identity function

$\lambda x.x$

Self-application function

$\lambda s.(s \ s)$

E.g. Apply identity function to self-application function

$$\blacktriangleright (\lambda x.x \ \lambda s.(s \ s)) = \lambda s.(s \ s)$$

E.g. Apply self-application function to identity function

$$\blacktriangleright (\lambda s.(s \ s) \ \lambda x.x) = (\lambda x.x \ \lambda x.x) = \lambda x.x$$

More functions

E.g. Apply self-application function to self-application function

$$(\lambda s.(s \ s) \ \lambda s.(s \ s))$$
$$\downarrow$$
$$(\lambda s.(s \ s) \ \lambda s.(s \ s))$$
$$\downarrow$$
$$\vdots$$
$$\downarrow$$
$$(\lambda s.(s \ s) \ \lambda s.(s \ s))$$

More functions

Function application function

$\lambda func.\lambda arg.(func\ arg)$

E.g. Apply function application function to identity function and self-application function

$((\lambda func.\lambda arg.(func\ arg)\ \lambda x.x)\ \lambda s.(s\ s))$

↓

$(\lambda arg.(\lambda x.x\ arg)\ \lambda s.(s\ s))$

↓

$(\lambda x.x\ \lambda s.(s\ s))$

↓

$\lambda s.(s\ s)$

α -conversion

Context of variables are local to their expressions, but having multiple variable of the same name may be confusing.

α -conversion is the renaming of variables.

Without α -conversion:

$$\begin{aligned} & (\lambda f. \lambda g. (\lambda f. (g \ f) \ f) \ \lambda p. \lambda q. p) \\ &= (\lambda f. \lambda g. (g \ f) \ \lambda p. \lambda q. p) \\ &= \lambda g. (g \ \lambda p. \lambda q. p) \end{aligned}$$

With α -conversion:

$$\begin{aligned} & (\lambda f. \lambda g. (\lambda f_1. (g \ f_1) \ f) \ \lambda p. \lambda q. p) \\ &= (\lambda f. \lambda g. (g \ f) \ \lambda p. \lambda q. p) \\ &= \lambda g. (g \ \lambda p. \lambda q. p) \end{aligned}$$

β -reduction example

$$\begin{aligned} & (((\lambda x. \lambda y. \lambda z. ((x \ y) \ z) \ \lambda f. \lambda a. (f \ a)) \ \lambda i. i) \ \lambda j. j) \\ &= ((\lambda y. \lambda z. ((\lambda f. \lambda a. (f \ a) \ y) \ z) \ \lambda i. i) \ \lambda j. j) \\ &= (\lambda z. ((\lambda f. \lambda a. (f \ a) \ \lambda i. i) \ z) \ \lambda j. j) \\ &= ((\lambda f. \lambda a. (f \ a) \ \lambda i. i) \ \lambda j. j) \\ &= (\lambda a. (\lambda i. i \ a) \ \lambda j. j) \\ &= (\lambda i. i \ \lambda j. j) \\ &= \lambda j. j \end{aligned}$$

Named Functions

Define named functions with: **def** < name > = < func > to make it easier to write out.

Some named functions:

```
def self_apply =  $\lambda s.(s \ s)$ 
```

```
def identity =  $\lambda x.x$ 
```

```
def apply =  $\lambda func.\lambda arg.(func \ arg)$ 
```

Named Functions

Let **def identity₂** = $\lambda x.((\text{apply } \text{identity}) \ x)$, then:

$$\begin{aligned}(\text{identity}_2 \ \text{identity}) &= (\lambda x.((\text{apply } \text{identity}) \ x) \ \text{identity}) \\&= ((\text{apply } \text{identity}) \ \text{identity}) \\&= ((\lambda f.\lambda a.(f \ a) \ \text{identity}) \ \text{identity}) \\&= ((\lambda a.(\text{identity } a) \ \text{identity}) \\&= (\text{identity } \text{identity}) \\&= (\lambda x.x \ \lambda x.x) \\&= \lambda x.x\end{aligned}$$

Named Functions

Example

Prove that the application function applies two arguments where the first is the function and the second its expression.

$$\begin{aligned} ((\text{apply } F) \text{ } AR) &= ((\lambda f. \lambda a. (f \text{ } a) \text{ } F) \text{ } AR) \\ &= (\lambda a. (F \text{ } a) \text{ } AR) \\ &= (F \text{ } AR) \end{aligned}$$

Named Functions

Example

Let **def self_apply₂** = $\lambda s.((\mathbf{apply} \ s) \ s)$. Prove that this named function is equivalent to the normal apply function.

$$\begin{aligned}(\mathit{self_apply}_2 \ z) &= (\lambda s.((\mathit{apply} \ s) \ s) \ z) \\&= ((\mathit{apply} \ z) \ z) \\&= ((\lambda f.\lambda a.(f \ a) \ z) \ z) \\&= (\lambda a.(z \ a) \ z) \\&= (z \ z)\end{aligned}$$

Named Functions

More named functions:

```
def select_first =  $\lambda$ first. $\lambda$ second.first
```

```
def select_second =  $\lambda$ first. $\lambda$ second.second
```

Select first example:

Notice how *identity* can be replaced with *A* and similarly you'd get *A* as result.

$$\begin{aligned}((\text{select_first identity}) \text{ apply}) &= ((\lambda \text{first} . \lambda \text{second} . \text{first identity}) \text{ apply}) \\ &= (\lambda \text{second} . \text{identity apply}) \\ &= \text{identity}\end{aligned}$$

Named Functions

Select second example:

$$\begin{aligned}((\text{select_second } A) \ B) &= ((\lambda \text{first}.\lambda \text{second}.\text{second } A) \ B) \\&= (\lambda \text{second}.\text{second } B) \\&= B\end{aligned}$$

Observe:

$$\begin{aligned}(\text{select_first identity}) &= (\lambda \text{first}.\lambda \text{second}.\text{first identity}) \\&= \lambda \text{second}.\text{identity} \\&= \lambda \text{second}.\lambda x.x && \text{(Renaming variables)} \\&= \lambda \text{first}.\lambda \text{second}.\text{second} \\&\equiv \text{select_second}\end{aligned}$$

Named Functions

$\text{def make_pair} = \lambda \text{first}.\lambda \text{second}.\lambda \text{func}.\text{((func first) second)}$

make_pair acts like a tuple.

$$\begin{aligned} & ((\text{make_pair } A) B) \\ &= ((\lambda \text{first}.\lambda \text{second}.\lambda \text{func}.\text{((func first) second) second}) A) B \\ &= (\lambda \text{second}.\lambda \text{func}.\text{((func A) second)}) B \\ &= \lambda \text{func}.\text{((func A) B)} \end{aligned}$$

Named Functions

Example application of *make_pair* result:

$$\begin{aligned} & (\lambda func. ((func\ A)\ B)\ select_first) \\ &= ((select_first\ A)\ B) \end{aligned}$$

Similarly:

$$\begin{aligned} & (\lambda func. ((func\ A)\ B)\ select_second) \\ &= ((select_second\ A)\ B) \end{aligned}$$

If Statement

We want `if` statements to be able to do more interesting things.

```
def cond =  $\lambda e_1. \lambda e_2. \lambda c ((c \ e_1) \ e_2)$ 
```

```
def true =  $\lambda a. \lambda b. a$ 
```

```
def false =  $\lambda a. \lambda b. b$ 
```

Notice how *true* and *false* come from *select_first* and *select_second*.

If Statement

Example

$$\begin{aligned} & (((cond\ x)\ y)\ true) \\ &= (((\lambda e_1. \lambda e_2. \lambda c. ((c\ e_1)\ e_2)\ x)\ y)\ true) \\ &= ((\lambda e_2. \lambda c. ((c\ x)\ e_2)\ y)\ true) \\ &= (\lambda c. ((c\ x)\ y)\ true) \\ &= ((true\ x)\ y) \\ &= ((\lambda a. \lambda b. a\ x)\ y) \\ &= (\lambda b. x\ y) \\ &= x \end{aligned}$$

Read like a ternary if statement, i.e.: $cond\ ?\ x : y$. In this case `true` is passed as the **condition**.

NOT Operator

Don't have operators yet so we can't get true or false out of conditions. Consider a truth table for NOT:

X	NOT X
T	F
F	T

Where T is true and F is false. How do you define a function for this?

NOT Operator

Solution lies in utilizing the ternary if statement: $x \text{ ? } \text{false} : \text{true}$.

def not = $\lambda x.((x \text{ false}) \text{ true})$

Derivation

$$\begin{aligned} & \lambda x.(((\text{cond } \text{false}) \text{ true}) \ x) \\ &= \lambda x.(((\lambda e_1.\lambda e_2.\lambda c.((c \ e_1) \ e_2) \ \text{false}) \ \text{true}) \ x) \\ &= \lambda x.((\lambda e_2.\lambda c.((c \ \text{false}) \ e_2) \ \text{true}) \ x) \\ &= \lambda x.(\lambda c.((c \ \text{false}) \ \text{true}) \ x) \\ &= \lambda x.((x \ \text{false}) \ \text{true}) \end{aligned}$$

Now we can pass x as an argument and get a result...

NOT Operator

Test the result

Passing true as argument:

$$\begin{aligned} & (\text{not } \text{true}) \\ &= (\lambda x. ((x \text{ false}) \text{ true}) \text{ true}) \\ &= ((\text{true false}) \text{ true}) \\ &= ((\lambda a. \lambda b. a \text{ false}) \text{ true}) && \text{(def. of true)} \\ &= (\lambda b. \text{false } \text{true}) \\ &= \text{false} \end{aligned}$$

AND Operator

Truth table for the AND operator:

X	Y	X AND Y
T	T	T
T	F	F
F	T	F
F	F	F

Where T is true and F is false. How do you define a function for this?

AND Operator

Comes from $x \text{ ? } y \text{ : false}$.

def and = $\lambda x. \lambda y. ((x \ y) \ \text{false})$

Derivation

$$\begin{aligned} & \lambda x. \lambda y. (((\text{cond } y) \ \text{false}) \ x) \\ &= \lambda x. \lambda y. ((((\lambda e_1. \lambda e_2. \lambda c. ((c \ e_1) \ e_2) \ y) \ \text{false}) \ x) \\ &= \lambda x. \lambda y. ((\lambda e_2. \lambda c. ((c \ y) \ e_2) \ \text{false}) \ x) \\ &= \lambda x. \lambda y. (\lambda c. ((c \ y) \ \text{false}) \ x) \\ &= \lambda x. \lambda y. ((x \ y) \ \text{false}) \end{aligned}$$

Now we can pass x and y as arguments and get a result...

AND Operator

Test the result

Passing true and false as arguments:

$$\begin{aligned} & ((\text{and } \text{true}) \text{ false}) \\ &= ((\lambda x. \lambda y. ((x \ y) \text{ false}) \text{ true}) \text{ false}) \\ &= (\lambda y. ((\text{true } y) \text{ false}) \text{ false}) \\ &= ((\text{true } \text{false}) \text{ false}) \\ &= ((\lambda a. \lambda b. a \ \text{false}) \text{ false}) && \text{(def. of true)} \\ &= (\lambda b. \text{false } \text{false}) \\ &= \text{false} \end{aligned}$$

AND Operator

Test the result

Passing true and true as arguments:

```
((and true) true)
= ((λx.λy.((x y) false) true) true)
= (λy.((true y) false) true)
= ((true true) false)
= ((λa.λb.a true) false)           (def. of true)
= (λb.true false)
= true
```

OR Operator

Truth table for the OR operator:

X	Y	X OR Y
T	T	T
T	F	T
F	T	T
F	F	F

Where T is true and F is false. How do you define a function for this?

OR Operator

Comes from $x \text{ ? } \text{true} : y$.

def or = $\lambda x. \lambda y. ((x \text{ true}) y)$

Derivation

$$\begin{aligned} & \lambda x. \lambda y. (((\text{cond } \text{true}) y) x) \\ &= \lambda x. \lambda y. (((\lambda e_1. \lambda e_2. \lambda c. ((c \text{ } e_1) e_2) \text{true}) y) x) \\ &= \lambda x. \lambda y. ((\lambda e_2. \lambda c. ((c \text{ true}) e_2) y) x) \\ &= \lambda x. \lambda y. (\lambda c. ((c \text{ true}) y) x) \\ &= \lambda x. \lambda y. ((x \text{ true}) y) \end{aligned}$$

Now we can pass x and y as arguments and get a result...

OR Operator

Test the result

Passing `true` and `false` as arguments:

```
((or true) false)
= ((λx.λy.((x true) y) true) false)
= (λy.((true true) y) false)
= ((true true) false)
= ((λa.λb.a true) false)      (def. of true)
= (λb.true false)
= true
```

Conditions & Operators

Exercise

Use Lambda Calculus to calculate the following:

$$\begin{aligned} & \text{if } ((T \vee F) \wedge (F \vee T)) \{ \\ & \quad T \wedge F \\ & \} \text{ else } \{ \\ & \quad T \\ & \} \end{aligned}$$

Conditions & Operators

Solution

Start with a basic outline:

$$(((cond\ x)\ y)\ z)$$

where z is the actual condition you're passing. Similar to the ternary if statement $z\ ?\ x\ :\ y$. Now:

$$x = ((and\ T)\ F),$$

$$y = T,$$

and

$$z = ((and\ ((or\ T)\ F))\ ((or\ F)\ T)).$$

Substituting yields:

$$(((cond\ ((and\ T)\ F))\ T)\ ((and\ ((or\ T)\ F))\ ((or\ F)\ T))) \quad (1)$$

Conditions & Operators

Solution continued

Breaking up the work into parts:

$$\begin{aligned}((or\ T)\ F) &= ((\lambda a.\lambda b.((a\ T)\ b)\ T)\ F) \\&= (\lambda b.((T\ T)\ b)\ F) \\&= ((T\ T)\ F) \\&= ((\lambda a.\lambda b.a\ T)\ F) \\&= (\lambda b.T\ F) \\&= T\end{aligned}\tag{2}$$

Conditions & Operators

Solution continued

$$\begin{aligned}((or\ F)\ T) &= ((\lambda a.\lambda b.((a\ T)\ b)\ F)\ T) \\&= (\lambda b.((F\ T)\ b)\ T) \\&= ((F\ T)\ T) \\&= ((\lambda a.\lambda b.b\ T)\ T) \\&= (\lambda b.b\ T) \\&= T\end{aligned}\tag{3}$$

Conditions & Operators

Solution continued

Now combining the results from (2) and (3):

$$\begin{aligned}((and\ T)\ T) &= ((\lambda x.\lambda y.((x\ y)\ F)\ T)\ T) \\&= (\lambda y.((T\ y)\ F)\ T) \\&= ((T\ T)\ F) \\&= ((\lambda a.\lambda b.a\ T)\ F) \\&= (\lambda b.T\ F) \\&= T\end{aligned}\tag{4}$$

Conditions & Operators

Solution continued

$$\begin{aligned}((and\ T)\ F) &= ((\lambda a.\lambda b.((a\ b)\ F)\ T)\ F) \\&= (\lambda b.((T\ b)\ F)\ F) \\&= ((T\ F)\ F) \\&= ((\lambda a.\lambda b.a\ F)\ F) \\&= (\lambda b.F\ F) \\&= F\end{aligned}\tag{5}$$

Conditions & Operators

Solution continued

Finally combining results from (4) and (5) into (1), yields:

$$\begin{aligned}(((cond\ F)\ T)\ T) &= (((\lambda e_1.\lambda e_2.\lambda c.((c\ e_1)\ e_2)\ F)\ T)\ T) \\&= ((\lambda e_2.\lambda c.((c\ F)\ e_2)\ T)\ T) \\&= (\lambda c.((c\ F)\ T)\ T) \\&= ((T\ F)\ T) \\&= ((\lambda a.\lambda b.a\ F)\ T) \\&= (\lambda b.F\ T) \\&= F\quad \square\end{aligned}\tag{6}$$

Numbers

How are numbers defined?

1 = successor of 0

2 = successor of 1

⋮

def zero = $\lambda x.x$

(identity function)

def succ = $\lambda n.\lambda s.((s \text{ false}) \ n)$

Now we can define **def one** = (**succ zero**) etc. for larger numbers.

Numbers

Numbers can be derived as follows:

$$\begin{aligned}one &= (\text{succ } zero) \\&= (\lambda n. \lambda s. ((s \text{ false}) \ n) \ zero) \\&= \lambda s. ((s \text{ false}) \ zero) \\two &= (\text{succ } one) \\&= (\lambda n. \lambda s. ((s \text{ false}) \ n) \ one) \\&= \lambda s. ((s \text{ false}) \ one) \\&= \lambda s. ((s \text{ false}) \ \lambda s. ((s \text{ false}) \ zero))\end{aligned}$$

Notice that in general any positive natural number ζ can be represented by:

$$\lambda s. ((s \text{ false}) \ \zeta - 1), \quad \ni \zeta \in \{one, two, \dots\}$$

Numbers

```
def is_zero = λn.(n  select_first)
```

Test

$$\begin{aligned}(is_zero \quad zero) &= (\lambda n.(n \quad select_first) \quad \lambda x.x) \\ &= (\lambda x.x \quad select_first) \\ &= select_first \\ &= true\end{aligned}$$

Numbers

Test

$$\begin{aligned} & (is_zero \ \lambda s. ((s \ false) \ \zeta - 1)) \\ &= (\lambda n. (n \ select_first) \ \lambda s. ((s \ false) \ \zeta - 1)) \\ &= (\lambda s. ((s \ false) \ \zeta - 1) \ select_first) \\ &= ((select_first \ false) \ \zeta - 1) \\ &= false \end{aligned}$$

\therefore `is_zero` is *false* for any number not zero.

Numbers

```
def pred1 = λn.(n  select_second)
```

Consider:

$$\begin{aligned}(\text{pred}_1 \text{ zero}) &= (\lambda n.(n \text{ select_second}) \ \lambda x.x) \\&= (\lambda x.x \ \text{select_second}) \\&= \text{select_second} \\&= \text{false}\end{aligned}$$

Which clearly isn't correct. So pred_1 doesn't work for zero .

Numbers

But what about other numbers? Let $\zeta = \lambda s.((s \text{ false}) \zeta - 1)$ be some positive natural number \therefore

$$\zeta \in \{\text{one}, \text{two}, \dots\}$$

Then:

$$\begin{aligned}(\text{pred}_1 \zeta) &= (\text{pred}_1 \lambda s.((s \text{ false}) \zeta - 1)) \\&= (\lambda n.(n \text{ select_second}) \lambda s.((s \text{ false}) \zeta - 1)) \\&= (\lambda s.((s \text{ false}) \zeta - 1) \text{ select_second}) \\&= ((\text{select_second false}) \zeta - 1) \\&= ((\lambda x.\lambda y.y \text{ false}) \zeta - 1) \\&= (\lambda y.y \zeta - 1) \\&= \zeta - 1\end{aligned}$$

Which is correct.

Numbers

To get around the problem with `zero` consider the function:

$$\lambda n.(((\text{cond } \text{zero}) (\text{pred}_1 \ n)) (\text{is_zero } n))$$

Which basically reads: `n == zero ? zero : n - 1`.

Simplifying:

$$\begin{aligned} & \lambda n.(((\text{cond } \text{zero}) (\text{pred}_1 \ n)) (\text{is_zero } n)) \\ &= \lambda n.(((\lambda e_1. \lambda e_2. \lambda c. ((c \ e_1) \ e_2) \ \text{zero}) (\text{pred}_1 \ n)) (\text{is_zero } n)) \\ &= \lambda n.((\lambda e_2. \lambda c. ((c \ \text{zero}) \ e_2) (\text{pred}_1 \ n)) (\text{is_zero } n)) \\ &= \lambda n.(\lambda c. ((c \ \text{zero}) (\text{pred}_1 \ n)) (\text{is_zero } n)) \\ &= \lambda n.(((\text{is_zero } n) \ \text{zero}) (\text{pred}_1 \ n)) \\ &= \lambda n.(((\text{is_zero } n) \ \text{zero}) (\lambda q. (q \ \text{select_second}) \ n)) \\ &= \lambda n.(((\text{is_zero } n) \ \text{zero}) (n \ \text{select_second})) \end{aligned}$$

Numbers

From the previous result we can now define:

def pred = $\lambda n.(((\text{is_zero } n) \text{ zero}) (n \text{ select_second}))$

Now: how to do addition? Consider:

$$\lambda x.\lambda y.(((\text{cond } x) ((\text{add } (\text{succ } x)) (\text{pred } y)))) (\text{is_zero } y)).$$

But this is defined recursively, which breaks Lambda Calculus.
Therefore...

Numbers

let rec add x y = if is_zero y then x else add (succ x) (pred y)