

# More Mathematical Tools

LING 255

February 5, 2008

## 1. CURRYING FUNCTIONS

Problem: given function application...

### (1) Function Application (Rule #1 of Interpretation)

If a node  $M$  has daughters  $D_1, D_2$  [in either order!], and  $D_1$  is a function that can take  $D_2$  as an argument, then  $\llbracket M \rrbracket = \llbracket D_1 \rrbracket(\llbracket D_2 \rrbracket)$

...we need a semantics for transitive verbs that doesn't require an ordered pair.

Currying<sup>1</sup> is a method of taking a function from pairs, and turning it into a function from one element of the pair to functions from the other element of the pair. Take, for example, the “+”

function, which takes a pair of numbers and returns a number: “+” = 
$$\left[ \begin{array}{l} \langle 1,1 \rangle \rightarrow 2 \\ \langle 1,2 \rangle \rightarrow 3 \\ \langle 2,1 \rangle \rightarrow 3 \\ \langle 3,2 \rangle \rightarrow 5 \\ \langle 2,2 \rangle \rightarrow 4 \\ \text{(etc. etc. etc.)} \end{array} \right]$$

Currying will give a new function  $f$  such that  $f(x)(y)$  returns what “+” would return when applied to  $\langle x, y \rangle$ . That means that, given  $x$ , it'll return a *function*. The functions are:

“+1” = 
$$\left[ \begin{array}{l} 1 \rightarrow 2 \\ 2 \rightarrow 3 \\ 3 \rightarrow 4 \\ \dots \end{array} \right]$$
      “+2” = 
$$\left[ \begin{array}{l} 1 \rightarrow 3 \\ 2 \rightarrow 4 \\ 3 \rightarrow 5 \\ \dots \end{array} \right]$$
      “+3” = 
$$\left[ \begin{array}{l} 1 \rightarrow 4 \\ 2 \rightarrow 5 \\ 3 \rightarrow 6 \\ \dots \end{array} \right]$$
      and so forth.

“Curried +” = 
$$\left[ \begin{array}{l} 1 \rightarrow [\text{the +1 function}] \\ 2 \rightarrow [\text{the +2 function}] \\ 3 \rightarrow [\text{the +3 function}] \\ \dots \end{array} \right] = \left[ \begin{array}{l} 1 \rightarrow \left[ \begin{array}{l} 1 \rightarrow 2 \\ 2 \rightarrow 3 \\ 3 \rightarrow 4 \\ \dots \end{array} \right] \\ 2 \rightarrow \left[ \begin{array}{l} 1 \rightarrow 3 \\ 2 \rightarrow 4 \\ 3 \rightarrow 5 \\ \dots \end{array} \right] \\ 3 \rightarrow \text{etc.} \\ \text{etc.} \end{array} \right]$$

---

<sup>1</sup> Named for Haskell Curry, but originally developed by Moses Schönfinkel; thus sometimes called *Schönfinkelization*.

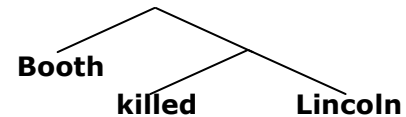
So our meaning for “killed” from last time:

- **[[killed]]** = that function from pairs of individuals to truth values such that, given an pair of individuals  $\langle x, y \rangle$ , it returns TRUE if and only if  $x$  killed  $y$

...can be Curried in this manner to give:

- **[[killed]]** = that function from individuals to [functions from individuals to truth values] such that, given an individual  $x$ , it returns [that function from individuals to truth values such that, given an individual  $y$ , it return TRUE if and only if  $y$  killed  $x$ ]

Now we need a structure in which **kill** can combine first with one name, and then the other. Fortunately, most syntacticians believe that the structure to the right is the one we want anyway.



*Digression: why is that?*

- It's much easier to conjoin **killed Lincoln** with other verb-object pairs than to conjoin **Booth killed** with other subject-verb pairs. That is, it's pretty natural to say **Booth [[killed Lincoln] and [fled the scene]]** than **[[Mary Todd loved] and [Booth killed]] Lincoln**. Not impossible, but harder.
- If someone asks, **What did Booth do?**, you could answer, **Shot Lincoln**. But if someone asks, **What happened to Lincoln?**, you can't just say **Booth shot**. (Similarly: **What Booth did was shot Lincoln** is OK, but not **What happened to Lincoln was Booth shot**.)
- And so on, through various other tests.

Now, to check that our meaning is right, we should go through a derivation, starting by combining **killed** with **Lincoln**. **killed** means “that function from...”

## 2. THE LAMBDA CALCULUS

Rather than write these things out in words, semanticists use a particular notation involving lambdas, where *lambda* ( $\lambda$ ) indicates “this thing is a function.”

The general format of a function in  $\lambda$ -notation is  $[\lambda A : B . C]$ . In this:

A is called the **argument variable**

(a letter standing for an arbitrary argument),

B is called the **domain condition**, introduced by a colon

(which puts a condition on the possible values of the argument variable),

C is called the **value description**, introduced by a period

(which gives the value assigned by the function)

For example: the “+1” function from the last section, which we’d generally have to call “that function from numbers to numbers which, given a particular number, returns the result of adding one to that number”. can be written  $[\lambda x : x \text{ is a number} . x + 1]$ . That is:

that function...	from numbers...	to numbers, which...
given a particular...	number, returns...	...the result of adding one to that number
$[\lambda x$	<b>: <math>x</math> is a number</b>	<b>. <math>x + 1]</math></b>

This notation expresses exactly what the wordy version does: the entire thing is a function, and the parts tell you that it takes a number, which we’ll call  $x$ , and returns the number  $x+1$ .

What about  $[[\text{killed}]]$ ?

- that function from individuals to [functions from individuals to truth values] such that, given an individual  $x$ , it returns [that function from individuals to truth values such that, given an individual  $y$ , it return TRUE if and only if  $y$  killed  $x$ ]

Taking it piece by piece:

that function...	from individuals...	to functions...	from individuals...	to truth values...
given an	individual $x$ , returns...	...given an	individual $y$ , returns...	...iff $y$ killed $x$
$[\lambda x$	: $x$ is an individual	. $[\lambda y$	: $y$ is an individual	. $y$ killed $x]$

- $[[\text{killed}]] = [\lambda x : x \text{ is an individual} . [\lambda y : y \text{ is an individual} . y \text{ killed } x]]$

(Note that “TRUE iff  $\alpha$ ” can be abbreviated to simply “ $\alpha$ ”.)

## 2.1. Semantic Types

If we want to talk about what type of object or function something is, we can also abbreviate “a function from individuals to functions from individuals to truth values”. Specifically:

- An individual has type  $e$ ;
- A truth value has type  $t$ ;
- A function from A’s to B’s has type  $\langle A, B \rangle$

For example:

- a function from individuals to individuals:  $\langle e, e \rangle$
- a function from individuals to truth values:  $\langle e, t \rangle$
- a function from (functions from individuals to truth values) to (functions from individuals to truth values):  $\langle \langle e, t \rangle, \langle e, t \rangle \rangle$

In general, the set of objects with type  $\sigma$  is written as  $D_\sigma$ . That means we can abbreviate:

- **[[killed]]** =  $[\lambda x : x \text{ is an individual} . [\lambda y : y \text{ is an individual} . y \text{ killed } x]]$

as

- **[[killed]]** =  $[\lambda x : x \in D_e . [\lambda y : y \in D_e . y \text{ killed } x]]^2$

## 2.2. *Lambda Conversion (aka lambda reduction)*

To apply a function written in  $\lambda$ -notation to its argument: remove the  $\lambda$  and its argument variable and domain condition; remove the argument; and substitute the argument wherever the argument variable occurs within the value description,

$$\begin{aligned} \llbracket \text{Lincoln snores} \rrbracket &= [\lambda x : x \in D_e . \text{TRUE iff } x \text{ snores}] (\text{Lincoln}) \\ &= [\cancel{\lambda x : x \in D_e} . \text{TRUE iff } x \text{ snores}] (\text{Lincoln}) \\ &= \text{TRUE iff Lincoln snores} \end{aligned}$$

or, for instance:

$$\begin{aligned} &[\lambda x . x^2 + 2x + 1](4) \\ &= [\cancel{\lambda x} . x^2 + 2x + 1](4) = 4^2 + 2 \cdot 4 + 1 = 25 \end{aligned}$$

If there are two arguments, the one closest to the  $\lambda$ -function is taken first. If there are two  $\lambda$ s, the one on the outside takes its argument first.

$$\begin{aligned} &[\lambda x : x \in D_e . [\lambda y : y \in D_e . \text{TRUE iff } y \text{ killed } x]] (\text{Lincoln})(\text{Booth}) \\ &= [\cancel{\lambda x : x \in D_e} . [\lambda y : y \in D_e . \text{TRUE iff } y \text{ killed } x]] (\text{Lincoln})(\text{Booth}) \\ &= [\lambda y : y \in D_e . \text{TRUE iff } y \text{ killed Lincoln}] (\text{Booth}) \\ &= [\cancel{\lambda y : y \in D_e} . \text{TRUE iff } y \text{ killed Lincoln}] (\text{Booth}) \\ &= \text{TRUE iff Booth killed Lincoln} \end{aligned}$$

## 2.3. *Important note #1: ordering and parentheses*

Note that, in  $[\lambda x : x \in D_e . [\lambda y : y \in D_e . \text{TRUE iff } y \text{ killed } x]] (\text{Lincoln})(\text{Booth})$ , the function takes Lincoln, and then takes Booth; Lincoln does not take Booth as an argument. That's fairly obvious in the case of Lincoln and Booth, who are both individuals, but you do have to be careful if one of the arguments is also a function.

<sup>2</sup> The notation  $x : x \in D_e$  is sometimes abbreviated further to  $x \in D_e$ .

For example:

$$(2) \quad [\lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)]](\lambda z : z \in D_n . z + 1)(5)$$

Here's the **wrong** way to start evaluating (2).

$$(3) \quad [\lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)]](\lambda z : z \in D_n . z + 1)(5) \\
 = [\lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)]](5 + 1) \\
 = [\lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)]](6)$$

Note that “6” is the wrong type of argument for the function. Here, we have:

$$f = \lambda z : z \in D_n . z + 1 \quad \langle n, n \rangle \\
 g = \lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)] \quad \langle \langle n, n \rangle, \langle n, n \rangle \rangle$$

And while (2) can be written as  $g(f)(x)$ , (3) is evaluating  $g(f(x))$ —not at all the same thing!

$$(4) \quad \text{What (3) actually evaluates: } g(f(x)) \\
 [\lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)]](\lambda z : z \in D_n . z + 1)(5)$$

The proper way to evaluate (2):

$$(5) \quad [\lambda y : y \in D_{\langle n, n \rangle} . [\lambda x : x \in D_n . y(2x)]](\lambda z : z \in D_n . z + 1)(5) \\
 = [\lambda x : x \in D_n . [\lambda z : z \in D_n . z + 1](2x)](5) \\
 = [\lambda x : x \in D_n . [\lambda z : z \in D_n . z + 1](2x)](5) \\
 = [\lambda x : x \in D_n . 2x + 1](5) \\
 = [\lambda x : x \in D_n . 2x + 1](5) \\
 = [2(5) + 1] \\
 = 11$$

## 2.4. Important note #2: alphabetic variants

Because variable names are arbitrary, it doesn't matter which ones we pick. That means that the following are all exactly identical:

- $[\lambda x : x \in D_e . [\lambda y : y \in D_e . \text{TRUE iff } y \text{ killed } x]]$
- $[\lambda y : y \in D_e . [\lambda x : x \in D_e . \text{TRUE iff } x \text{ killed } y]]$
- $[\lambda f : f \in D_e . [\lambda n : n \in D_e . \text{TRUE iff } n \text{ killed } f]]$
- $[\lambda x_1 : x_1 \in D_e . [\lambda x_2 : x_2 \in D_e . \text{TRUE iff } x_2 \text{ killed } x_1]]$

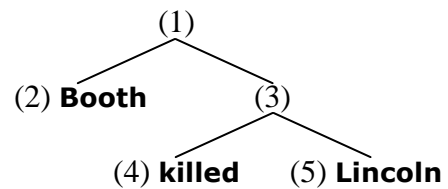
It doesn't matter at all what variable names we choose. (Conventionally, we use  $x, y, z$  as variables of type  $e$ , and  $P, Q$  as variables of type  $\langle e, t \rangle$ , but the domain restriction specifies the type of the variable, so in the end we can tell no matter what variable name is used.) The following, however, are *not* equivalent to the above function:

- $\lambda y : y \in D_e . \lambda x : x \in D_e . \text{TRUE iff } y \text{ killed } x$
- $\lambda x : x \in D_e . \lambda x : x \in D_e . \text{TRUE iff } x \text{ killed } x$

The replacement of variables must be consistent throughout; and cannot use variables already being used.

### 3. USING ALL OF THIS

When giving the meaning of an expression (i.e., its denotation), you'll want to work through it step by step, noting at each step what rule or rules are being used. This can actually be done "top-down" or "bottom-up". For the structure at the right, whose nodes have been conveniently numbered:



#### 3.1. Top-Down

[[1]]	= [[3]]([[2]])	Function Application (FA)
	= [[4]]([[5]])([[2]])	FA
	= [[ $\lambda y \in D_e . [\lambda x \in D_e . x \text{ killed } y]$ ]]([[Lincoln]])([[Booth]])	Lexical Meaning (LM)
	= [[ $\lambda y \in D_e . [\lambda x \in D_e . x \text{ killed } y]$ ]](Lincoln)(Booth)	LM $\times$ 2
	= [ $\lambda x_e . x \text{ killed Lincoln}$ ](Booth)	$\lambda$ -conversion
	= TRUE iff Booth killed Lincoln	$\lambda$ -conversion

Note: the order of operations doesn't matter. The key is that each line is equivalent to the one above it by substitution of an expression for an equivalent one.

#### 3.2. Bottom-Up

[[5]]	= Lincoln	LM
[[4]]	= [ $\lambda y \in D_e . [\lambda x \in D_e . x \text{ killed } y]$ ]	LM
[[3]]	= [[4]]([[5]])	FA
	= [ $\lambda y \in D_e . [\lambda x \in D_e . x \text{ killed } y]$ ](Lincoln)	steps 1 and 2
	= [ $\lambda x_e . x \text{ killed Lincoln}$ ]	$\lambda$ -conversion
[[2]]	= Booth	LM
[[1]]	= [[3]]([[2]])	FA
	= [ $\lambda x_e . x \text{ killed Lincoln}$ ](Booth)	steps 3 and 4
	= TRUE iff Booth killed Lincoln	$\lambda$ -conversion